2

WORKING MATERIAL

FOR THE LECTURES OF

Helmut Schwichtenberg

COURSE ON NORMALIZATION

INTERNATIONAL SUMMER SCHOOL

ON

LOGIC, ALGEBRA AND COMPUTATION

MARKTOBERDORF, GERMANY, JULY 25 - AUGUST 6, 1989

THIS SUMMER SCHOOL IS ORGANIZED UNDER THE AUSPICES OF THE TECHNISCHE UNIVERSITÄT MÜNCHEN AND IS SPONSORED BY THE NATO SCIENCE COMMITTEE AS PART OF THE 1989 ADVANCED STUDY INSTITUTES PROGRAMME. PARTIAL SUPPORT FOR THE CONFERENCE WAS PROVIDED BY THE EUROPEAN RESEARCH OFFICE AND THE NATIONAL SCIENCE FOUNDATION AND BY VARIOUS INDUSTRIAL COMPANIES.

Normalization

Helmut Schwichtenberg[1]

First draft of July 17, 1989

[1]Mathematisches Institut der Universität München, Theresienstraße 39, D-8000 München 2, E-Mail: UG143CA/DM0LRZ01/BITNET

Abstract

The aim of this course is to present a basic technique from proof theory, Gentzen's normalization for natural deduction systems, and to discuss some of its applications.

By normalization we mean a collection of algorithms transforming a given derivation into a certain normal form. A derivation is called *normal* if it does not contain any detour or — in Gentzen's terminology — *cut*, i. e. an application of an introduction rule immediately followed by an application of an elimination rule. Such normalization algorithms are useful because they allow to "straighten out" complex derivations and in this way extract hidden information.

We will treat many applications which demonstrate this, e. g. the subformula principle, Herbrand's theorem, the interpolation theorem, and an exact characterization of the initial cases of transfinite induction provable in arithmetic.

From the computer science point of view, an even more interesting field of application for normalization algorithms is the possiblility to extract the constructive content of a maybe complex mathematical argument. Such algorithms can yield verified programs from derivations proving that certain specifications can be fulfilled. Of course, the feasability of programs obtained in this way will depend to a large extent on a good choice of the derivation, which should be done on the basis of a good idea for an algorithm. However, in this approach it is possible to use ordinary mathematical machinery for the development of programs. (KR)

Limitations in time have kept me from including more material into these notes. In particular, I have left out the subject of codes for infinitary derivations and their use to extract bounds from proofs of classical existence theorems (cf. [Sch77b,Sch86]). I also had to leave out a planned chapter on normalization for type theory; part of what I wanted to say is contained in [Sch88].

Contents

Chapter 1

Normalization for propositional logic

We begin with the simplest possible logical system, the implicational fragment of propositional calculus. Only two logical rules are present, allowing introduction and elimination of implication[1]. By the so-called Curry-Howard isomorphism this system is the same as the pure typed λ-calculus, and we shall make extensive use of that fact.

The restriction to the implicational fragment of propositional logic is not essential since it is well-known that full classical propositional logic can be embedded in it. In Section 1.1 we will discuss this embedding.

1.1 Pure implicational logic as a typed λ-calculus

Formulas are built up from propositional variables denoted by P, Q by means of $(A \to B)$. [2] Derivations are built up from assumption variables x^A, y^B by means of the rule $\to^+$ of implication introduction (or λ-abstraction)

$$(\lambda x^A r^B)^{A \to B}$$

and the rule $\to^-$ of implication elimination (or application)

$$(t^{A \to B} s^A)^B.$$

[1]This means that we treat so-called minimal logic, formulated in Gentzen's natural deduction style.

[2]We have chosen the unary implication $A \to B$ instead of $A_1, \dots, A_m \to B$ just for notational convenience.

A derivation r^B whose free assumption variables are $x_1^{A_1}, \ldots, x_m^{A_m}$ is also called a derivation of B from $A_1, \ldots, A_m$. For readability we often leave out formula superscripts when they are obvious from the context or non-essential.

As an example we give derivations of the two Frege formulas

$$A \to (B \to A) \tag{1.1}$$

$$A \to (B \to C)) \to ((A \to B) \to (A \to C)) \tag{1.2}$$

A derivation of 1.1 is

$$\lambda x^A(\lambda y^B x^A),$$

and a derivation of 1.2 is

$$\lambda u^{A\to(B\to C)} \lambda v^{A\to B} \lambda w^A((uw)(vw)).$$

Both derivations can be easily written in the more usual tree form. For obvious reasons we will also use the word term for derivations and type for formulas. The possibility to treat derivations as terms and formulas as types has been discovered by H. B. Curry and elaborated by W. A. Howard in [How80a]. This correspondence can easily be shown to be an isomorphism; it is called the Curry-Howard-isomorphism.

Note that our *pure implicational* logic contains full classical propositional logic, as follows. Choose a particular propositional variable and denote it by $\perp$ (falsity). Associate with any formula A in the language of classical propositional logic a finite list $\vec{A}$ of formulas in our implicational language, by induction on A [3]:

$$\begin{array}{rcl}
P & \mapsto & P \\
\neg A & \mapsto & \vec{A} \to \perp \\
A \to B & \mapsto & \vec{A} \to B_1, \ldots, \vec{A} \to B_n \\
A \wedge B & \mapsto & \vec{A}, \vec{B} \\
A \vee B & \mapsto & (\vec{A} \to \perp), (\vec{B} \to \perp) \to \perp
\end{array}$$

Then, if A is a formula in the language of full classical propositional logic and $A_1, \ldots, A_m$ is its associated list, A is derivable in classical propositional logic iff each A_i is derivable in pure implicational logic from stability assumptions $\neg\neg P \to P$ (with $\neg B$ denoting $B \to \perp$) for all propositional variables P in A.

[3] $A_1, \ldots, A_m \to B$ is supposed to mean $(A_1 \to (A_2 \to \ldots (A_m \to B) \ldots))$

1.2 Conversion

We are interested in the following process of simplification of terms:

$$(\lambda\vec{x}x.r)\vec{s}s \text{ converts into } (\lambda\vec{x}.r_x[s])\vec{s}.$$

Here $\vec{x}$ and $\vec{s}$ denote finite lists $x_1 \dots x_m$ and $s_1 \dots s_m$, and $\lambda\vec{x}x.r$ denotes $\lambda x_1 \dots \lambda x_m \lambda x r$. Terms of the form $(\lambda\vec{x}x.r)\vec{s}s$ are called *convertible* [4]. We write

$$r \to r'$$

if r' is obtained from r as follows. Mark some occurences of convertible subterms in r. Then convert them all simultaneously [5].

More precisely, $r \to r'$ is defined by the following rules

1. $x \to x$,
2. $r \to r'$ implies $\lambda x r \to \lambda x r'$,
3. $r \to r'$ and $s \to s'$ imply $rs \to r's'$,
4. $r \to r'$ and $\vec{s} \to \vec{s'}$ and $s \to s'$ imply $(\lambda\vec{x}x.r)\vec{s}s \to (\lambda\vec{x}.r'_x[s'])\vec{s'}$.

As a special case, we take

$$r \xrightarrow{1} r'$$

to mean that r' is obtained from r by converting exactly one convertible subterm in r. Finally

$$r \to^* r' \qquad (r \text{ reduces to } r')$$

[4]Note that converting $(\lambda\vec{x}x.r)\vec{s}s$ into $(\lambda\vec{x}.r_x[s])\vec{s}$ may be viewed as first converting $(\lambda\vec{x}x.r)\vec{s}s$ "permutatively" into $(\lambda\vec{x}(\lambda x r)s)\vec{s}$ and then performing the inner conversion to obtain $(\lambda\vec{x}.r_x[s])\vec{s}$. One may ask why we take this conversion relation as our basis and not the more common $(\lambda x r)s \mapsto r_x[s]$. The reason is that our notion of level is defined with the clause level$(A \to B) = \max($level $A)+1,$ level $B)$ and not $= \max($level $A,$ level $B)+1$; this in turn seems reasonable since then the level of $P_1, \dots P_m \to Q$ (i. e. of $(P_1 \to (P_2 \to \dots (P_m \to Q)\dots))$) is 1 and hence independent of m. But given this definition of level, and given the need in some arguments (e. g. in Theorem 1.3.1) to perform conversions of highest level first, we must be able to convert $(\lambda xy.r)st$ with x of a low and y of a high level into $(\lambda x.r_y[t])s$. —In addition, since we allow more conversions here, the results on strong normalization and upper bounds for the length of arbitrary reduction sequences get stronger.

[5]Hence new convertible subterms generated by such a conversion can *not* be converted.

denotes the transitive and reflexive closure of $\to$ (or equivalently of $\xrightarrow{1}$).

A term is said to be in *normal form* if it does not contain a convertible subterm.

We want to show now that any term reduces to a normal form. This can be seen easily if we follow a certain order in our conversions. To define this order we have to make use of the fact that all our terms (i.e. derivations) have types (i.e. formulas).

Define the *level* of a formula by

$$\text{level } P = 0,$$

$$\text{level}(A \to B) = \max((\text{level } A) + 1, \text{level } B).$$

A convertible derivation

$$(\lambda \vec{x}^{\vec{A}} x^A . r)\vec{s}s$$

is also called a *cut* with *cut-formula* A. By the level of a cut we mean the level of its cut-formula. The *cut-rank* of a derivation r is the least number bigger than the levels of all cuts in r. Now let t be a derivation of cut-rank $k+1$. Pick a cut

$$(\lambda \vec{x}x.r)\vec{s}s$$

of the maximal level k in t, such that s does not contain another cut of level k. (E.g., pick the rightmost cut of level k.) Then it is easy to see that replacing the picked occurrence of $(\lambda \vec{x}x.r)\vec{s}s$ in t by $(\lambda \vec{x}.r_x[s])\vec{s}$ reduces the number of cuts of the maximal level k in t by 1. Hence

Theorem 1.2.1 *We have an algorithm which reduces any given term into a normal form.* □

We now want to give an estimate of the number of conversion steps our algorithm takes until it reaches the normal form. The key observation for this estimate is the obvious fact that replacing one occurrence of

$$(\lambda \vec{x}x.r)\vec{s}s \qquad \text{by} \qquad (\lambda \vec{x}.r_x[s])\vec{s}$$

in a given term t at most squares the length of t; here the *length* of t is taken to be the number of variables in t (except those immediately following a λ-symbol).

A bound $s_k l$ for the number of steps our algorithm takes to reduce the rank of a given term of length l by k can be derived inductively, as follows. Let $s_0 l := 0$. To obtain $s_{k+1} l$, first note that by induction hypothesis it takes

$\leq s_k l$ steps to reduce the rank by k. The length of the resulting term is $\leq l^{2^s}$ where $s := s_k l$ since any step (i.e. conversion) at most squares the length. Now to reduce the rank by one more the number of additional steps is obviously bounded by that length. Hence the total number of steps to reduce the rank by $k+1$ is bounded by

$$s_k l + l^{2^{s_k l}} =: s_{k+1} l.$$

Theorem 1.2.2 (Upper bound for the complexity of normalization) *The normalization algorithm given in the proof of Theorem 1.2.1 takes at most $s_k l$ steps to reduce a given term of cut-rank k and length l to normal form, where*

$$s_0 l := 0 \textit{ and } s_{k+1} l := s_k l + l^{2^{s_k l}}. \square$$

1.3 Uniqueness

We shall show that the normal form of a term is uniquely determined [6]. The main idea of the proof (due to J. B. Rosser and W. W. Tait) is to use the relation $r \rightarrow r'$ defined in section 1.2. Its crucial property is given by

Lemma 1.3.1

$$\left.\begin{array}{l} r \rightarrow r' \\ t \rightarrow t' \end{array}\right\} \Rightarrow r_y[t] \rightarrow r'_y[t']$$

The proof is by induction on the definition of $r \rightarrow r'$. All cases are obvious exept possibly rule 4. So assume $r \rightarrow r'$, $\vec{s} \rightarrow \vec{s'}$ and $s \rightarrow s'$. Then

$$r_y[t] \rightarrow r'_y[t'], \qquad \vec{s}_y[t] \rightarrow \vec{s'}_y[t'] \text{ and } s_y[t] \rightarrow s'_y[t']$$

by induction hypothesis, and hence

$$\underbrace{(\lambda\vec{x}x.r_y[t])\vec{s}_y[t]s_y[t]}_{((\lambda\vec{x}x.r)\vec{s}s)_y[t]} \rightarrow \underbrace{(\lambda\vec{x}.(r'_y[t'])_x[s'_y[t']])\vec{s'}_y[t']}_{((\lambda\vec{x}.r'_x[s'])\vec{s'})_y[t']}$$

by definition of $\rightarrow$. $\square$

Lemma 1.3.2 *Assume $r \rightarrow r'$ and $r \rightarrow r''$. Then we can find a term r''' such that $r' \rightarrow r'''$ and $r'' \rightarrow r'''$.*

[6]The argument in this section also applies to type-free terms, i.e. terms without formula superscripts.

The proof is by induction on the definition of $r \to r'$. Again all cases are obvious exept possibly the situation where either $r \to r'$ or $r \to r''$ is obtained via rule 4. By symmetry we may assume the former. But then the claim follows from Lemma 1.3.1: If

$$(\lambda \vec{x} x.r)\vec{s}s \to (\lambda \vec{x}.r'_x[s'])\vec{s'}$$

and

$$(\lambda \vec{x} x.r)\vec{s}s \to (\lambda \vec{x} x.r'')\vec{s''}s'',$$

then

$$(\lambda \vec{x} x.r)\vec{s}s \to (\lambda \vec{x}.r'_x[s'])\vec{s'} \to (\lambda \vec{x}.r'''_x[s'''])\vec{s'''}$$

and

$$(\lambda \vec{x} x.r)\vec{s}s \to (\lambda \vec{x} x.r'')\vec{s''}s'' \to (\lambda \vec{x}.r'''_x[s'''])\vec{s'''},$$

and if

$$(\lambda \vec{x} x \vec{y} y.r)\vec{s}s\vec{t}t \to (\lambda \vec{x}\vec{y} y.r'_x[s'])\vec{s'}\vec{t'}t'$$

and

$$(\lambda \vec{x} x \vec{y} y.r)\vec{s}s\vec{t}t \to (\lambda \vec{x} x \vec{y}.r''_y[t''])\vec{s''}s''\vec{t''}$$

then

$$(\lambda \vec{x} x \vec{y} y.r)\vec{s}s\vec{t}t \to (\lambda \vec{x}\vec{y} y.r'_x[s'])\vec{s'}\vec{t'}t' \to (\lambda \vec{x}\vec{y}.r'''_{x,y}[s''', t'''])\vec{s'''}\vec{t'''}$$

and

$$(\lambda \vec{x} x \vec{y} y.r)\vec{s}s\vec{t}t \to (\lambda \vec{x} x \vec{y}.r''_y[t''])\vec{s''}s''\vec{t''} \to (\lambda \vec{x}\vec{y}.r'''_{x,y}[s''', t'''])\vec{s'''}\vec{t'''}$$

Theorem 1.3.1 (Church-Rosser) *Assume $r \to^* r'$ and $r \to^* r''$. Then we can find a term r''' such that $r' \to^* r'''$ and $r'' \to^* r'''$.*

The proof is immediate from Lemma 1.3.2. □

Corollary 1.3.2 *Assume $r \to^* r'$ and $r \to^* r''$, where both r' and r'' are in normal form. Then r' and r'' are identical.* □

1.4 A lower bound for the complexity of normalization

In Theorem 1.2.2 we have obtained an upper bound on the number of conversion steps our particular normalization algorithm of Theorem 1.2.1 takes to reach the normal form. This upper bound was superexponential in the length of the given term. It is tempting to think that by choosing a clever normalization strategy one might be able to reduce that bound significantly. It is the purpose of the present section to show that this is impossible. More precisely, we will construct terms r_n of length$3n$ and show that any normalization algorithm needs at least $2_{n-2} - n$ conversions (with $2_0 := 1, 2_{n+1} := 2^{2_n}$) to reduce r_n to its normal form.

The fact that there is no elementary algorithm (i. e. whose time is exponentially bounded) to compute the normal form of terms also follows from Statman [Sta79], where it is shown more generally that the problem whether two terms r_1 and r_2 have the same normal form is not elementary recursive. The simple example treated here is taken from [Sch82, p. 455]

The pure types k are defined inductively by $0 := P$ (some fixed propositional variable) and $k + 1 = k \to k$. We define iteration terms I_n of pure type $k + 2$ by

$$I_n :\equiv \lambda f \lambda x f(f(\dots f(fx)\dots)),$$

with n occurrences of f after $\lambda f \lambda x$; here f, x are variables of type $k + 1, k$, respectively. Let $f \circ g$ be an abbreviation for $\lambda x f(gx)$, and let $r = s$ mean that r and s have the same normal form. With this notation we can write

$$I_n = \lambda f \underbrace{f \circ f \circ \dots \circ f}_{n}$$

The main point of our argument is the following simple lemma, which can be traced back to Rosser (cf. [Chu41, p. 30])

Lemma 1.4.1 *1.* $(I_m f) \circ (I_n f) = I_{m+n} f$

2. $I_m \circ I_n = I_{m \cdot n}$

3. $I_m I_n = I_{n^m}$ □

As an immediate consequence we have

$$r_n :\equiv \underbrace{I_2 I_2 \dots I_2}_{n} = I_{2_n}$$

Now consider any sequence of reduction steps transforming r_n into its normal form, and let s_n denote the total number of reduction steps in this sequence.

Theorem 1.4.1 $s_n \geq 2_{n-2} - n$

Proof: The length of r_n is $3n$. Note that any conversion step can at most square the length of the original term. Hence we have

$$\begin{aligned} 2_n &< \text{length of } I_{2_n} \qquad (\text{the normal form of } r_n) \\ &\leq (\text{length of } r_n)^{2^{s_n}} \\ &= (3n)^{2^{s_n}} \\ &\leq 2^{2^{n+s_n}} \qquad (\text{since } 3n \leq 2^{2^n}), \end{aligned}$$

and the theorem is proved. □

1.5 Strong normalization

In Section 1.2 we have proved that any term can be reduced to a normal form, and in Section 1.3 we have seen that this normal form is uniquely determined. But it is still conceivable that there might be an odd reduction sequence which does not terminate at all. It is the aim of the present section to show that this is impossible. This fact is called the strong normalization theorem.

For the proof we employ a powerful method due to W.W. Tait, which is based on so–called strong computability predicates. These are defined by induction on the types (i. e. formulas) as follows.

A term r^A with A of level 0 (i. e. a propositional variable) is strongly computable iff r is strongly normalizable, i. e. every reduction sequence starting from r terminates. A term $r^{A \to B}$ is strongly computable iff for all strongly computable s^A also $(rs)^B$ is strongly computable.

A term r is ***strongly computable under substitution*** iff for all strongly computable $\vec{s}$ the result of substituting $\vec{s}$ for all variables free in r is again strongly computable.

Lemma 1.5.1 *Let A be a formula.*

1. *Any strongly computable term r^A is strongly normalizable.*

2. *x^A is strongly computable.*

We prove 1 and 2 simultaneously by induction on A. For A of level 0 both claims are obvious. Now consider $A \to B$. For 1, assume that $r^{A \to B}$ is strongly computable. By induction hypothesis 2 and the definition of strong computability we know that $(rx)^B$ is strongly computable and hence that any reduction sequence starting with rx terminates (by induction hypothesis 1). But this obviously implies that the same is true for r. For 2, assume that $\vec{r}$ are strongly computable. We have to show that $x\vec{r}$ (which is to be of level 0) is strongly computable, i. e. that any reduction sequence starting with $x\vec{r}$ terminates. But this follows from induction hypothesis 1, which says that any reduction sequence starting from r_i terminates. □

Lemma 1.5.2 *If $r \xrightarrow{1} r'$ and r is strongly computable, then r' is strongly computable.*

Proof: Let $\vec{s}$ be strongly computable. We have to show that $r'\vec{s}$ is strongly computable, i. e. that any reduction sequence starting from $r'\vec{s}$ terminates. But this is obviously true, because otherwise we would also have an infinite reduction sequence for $r\vec{s}$. □

Lemma 1.5.3 *Any term r is strongly computable under substitution.*

The proof is by induction on the height of r. $\boxed{x}$ obvious. $\boxed{rs}$ Let $\vec{t}$ be strongly computable. We have to show that $r[\vec{t}]s[\vec{t}]$ is strongly computable. But this holds, since by induction hypothesis we know that $r[\vec{t}]$ as well as $s[\vec{t}]$ are strongly computable. $\boxed{\lambda\vec{x}x.r}$ Let $\vec{t}$ be strongly computable. We have to show that $\lambda\vec{x}x.r[\vec{t}]$ is strongly computable. So let $\vec{s}, s$ and $\vec{r}$ be strongly computable. We must show that $(\lambda\vec{x}x.r[\vec{t}])\vec{s}s\vec{r}$ is strongly computable, i. e. than any reduction sequence for it terminates. So assume we have an infinite reduction sequence. Since $r[\vec{t}], \vec{s}, s$ and $\vec{r}$ all are strongly normalizable, there must be a term $(\lambda\vec{x}x.r[\vec{t}]')\vec{s'}s'\vec{r'}$ with $r[\vec{t}] \to^* r[\vec{t}]', \vec{s'} \to^* \vec{s}, s \to^* s'$ and $\vec{r} \to^* \vec{r'}$ in that reduction sequence where a "head conversion" is applied, which we may assume to yield

$$(\lambda\vec{x}.(r[\vec{t}]')[s'])\vec{s'}\vec{r'}.$$

But $r[\vec{t}] \to^* r[\vec{t}]'$ implies $\lambda\vec{x}.r[s,\vec{t}] \to^* \lambda\vec{x}.(r[\vec{t}]')[s']$, and hence the fact that $\lambda\vec{x}.r$ is (by induction hypothesis) strongly computable under substitution together with Lemma 1.5.2 implies that $(\lambda\vec{x}.(r[\vec{t}]')[\vec{s'}]$ is strongly computable. But then, again by Lemma 1.5.2, also $(\lambda\vec{x}.(r[\vec{t}])'[s'])\vec{s'}\vec{r'}$ is strongly computable and therefore strongly normalizable. This contradicts our assumption above that we have an infinite reduction sequence. □

From Lemma 1.5.3 and both parts of Lemma 1.5.1 we can conclude immediately

Theorem 1.5.1 *Any term r is strongly normalizable.* □

1.6 An upper bound for the length of arbitrary reduction sequences

By section 1.5 we already know that the full reduction tree for a given term is finite; hence its height bounds the length of any reduction sequence. However, it is not obvious how a reasonable estimate for that height might be obtained.

Here we note that a much simpler tree (to be called below 0-tree) which is essentially the head reduction sequence (and branches only in the case $xt_1 \dots t_n$ with successors $t_1\vec{y}_1, \dots, t_n\vec{y}_n$) has the property that the number of its nodes with conversions bounds the length of any reduction sequence[7]. The height of that tree, and hence also the number of its nodes, can be estimated using a technique due to Howard [How80b], which in turn is based on work of Sanchis [San67] and Diller [Dil68]. This gives the desired upper bound.

The method of Gandy [Gan80b] can also be used to obtain a bound for the length of arbitrary reduction sequences; this is carried out in [Sch82]. However, the bound derived here, apart from being more intelligible, is also better.

For any term r of level 0 a head reduction tree of degree k (to be called k-tree) for r is generated by the following rules. All terms in that tree are supposed to have level 0.

1. $(\lambda x r)s\vec{t}$ has successor $r_x[s]\vec{t}$.
2. $xt_1 \dots t_n$ has successors $t_1\vec{y}_1, \dots, t_n\vec{y}_n$. In particular, x of level 0 has no successor.
3. $rt_1 \dots t_n$ with r level $\leq k$ and $n > 0$ has successor $r\vec{y}, t_1\vec{y}_1, \dots, t_n\vec{y}_n$.

Note that for $(\lambda x r)s\vec{t}$ with x of level $\geq k$ only rule 1 (and not rule 3) can be applied when constructing a k-tree.

Hence the relation $|r|_k \leq a$ for terms r of level 0 (to be read: r has a k-tree of height $\leq a$) can be defined inductively, as follows:

[7]This is not quite true, but only for so-called λ-I-terms, where any variable bound by λ actually occurs in the kernel. But the general case can be easily reduced to this one by introducing dummy variables; this is carried out below.

1. If $|r_x[s]\vec{t}|_k \le a_0 < a$, then $|(\lambda xr)s\vec{t}|_k \le a$.
2. If $|t_i\vec{y}_i|_k \le a_i < a$ for $i = 1, \dots, n$, then $|x\vec{t}|_k \le a$. In particular, $|x|_k \le a$ for any a, for x of level 0.
3. If $|r\vec{y}|_k \le a_0 < a$ and $|t_i\vec{y}_i|_k \le a_i < a$ for $i = 1, \dots, n$, and if r has level $\le k$, then $|r\vec{t}|_k \le a$.

Lemma 1.6.1 *If $|r|_k \le a$ and $|s_j\vec{y}_j|_k \le b$ for $j = 1, \dots, m$, and if $s_1, \dots, s_m$ have levels $\le k$, then $|r_{\vec{x}}[\vec{s}]|_k \le b + a$.*

The proof is by induction on the generation of $|r|_k \le a$. We write t^* for $t_{\vec{x}}[\vec{s}]$.

1. $|r^*_x[s^*]\vec{t}^*|_k \le b + a_0 < b + a$ by induction hypothesis, and hence we have $|(\lambda xr^*)s^*\vec{t}^*|_k \le b + a$.
2. $|t^*_i\vec{y}_i|_k \le b + a_i < b + a$ for $i = 1, \dots, m$ by induction hypothesis, and hence $|x\vec{t}^*|_k \le b + a$. If x is one of the x_j to be substituted by s_j, we have to use rule 3 instead. This can be done since s_j has level $\le k$ and by assumption $|s_j\vec{y}_j|_k \le b$. Now in case a is not zero we have $b < b + a$ and hence $|s_j\vec{t}^*|_k \le b + a$ by rule 3. In case a is zero there are no t_i's and we have used rule 2 to generate $|x|_k \le 0$. Then $|s_j|_k \le b + 0$ holds by assumption.
3. $|r^*\vec{y}|_k \le b + a_0 < b + a$ and $|t^*_i\vec{y}_i|_k \le b + a_i < b + a$ hold by induction hypothesis, and hence $|r^*\vec{t}^*| \le b + a$. □

Lemma 1.6.2 *If $|r|_{k+1} \le a$, then $|r|_k \le 2^a$.*

The proof is by induction on the generation of $|r|_{k+1} \le a$. The only case which requires some attention is when $|r\vec{t}|_{k+1} \le a$ was generated from $|r\vec{y}|_{k+1} \le a_0 < a$ and $|t_i\vec{y}_i|_{k+1} \le a_i < a$ for $i = 1, \dots, n$ by rule 3, and r has level $k + 1$. By induction hypothesis we then have $|r\vec{y}|_k \le 2^{a_0}$ and $|t_i\vec{y}_i|_k \le 2^{a_i}$ for $i = 1, \dots, n$. Now by the previous Lemma we can conclude that $|r\vec{t}|_k \le 2^{\max(a_1,\dots,a_n)} + 2^{a_0} \le 2^a$. □

It now follows that from a bound for the height of some k-tree for r (and for sufficiently big k such a bound can be obtained easily; see Lemma 1.6.5 below) we can derive a bound for the height of the 0-tree of r. Note that this 0-tree is uniquely determined, since only rules 1 and 2 can be applied; we shall call it the head reduction tree for r and write $|r|_0$ for its height, i. e. for the least a such that $|r|_0 \le a$.

Our key observation is that, under a slight additional hypothesis, the number $\#r$ of conversions in the head reduction tree of r bounds the length of any reduction sequence. More precisely, $\#r$ is defined for any term r of level 0 by induction on the generation of $|r|_0 \leq a$ by the rules above, i. e.

1. $\#(\lambda xr)s\vec{t} := (\#r_x[s]\vec{t}) + 1$,
2. $\#xt_1 \dots t_n := \sum_{i=1}^{n} \#t_i\vec{y}_i$.

Note that from 1. we can conclude

$$\#(\lambda\vec{x}x.r)\vec{s}s\vec{t} = (\#(\lambda\vec{x}x.r_x[s])\vec{s}\vec{t} + 1;$$

this will be used below.

A term is called a λ-I-term if in its generation a subterm λxr is formed only in case $x \in FVr$.

Lemma 1.6.3 *Let r be a λ-I-term of level* 0. *Then $r \xrightarrow{1} r'$ implies that $\#r > \#r'$.*

Proof. We show more generally that for r a λ-I-term with $z \in FVr$ we have

$$\#r_z[(\lambda\vec{x}x.p)\vec{q}q] > \#r_z[(\lambda\vec{x}.p_x[q])\vec{q}]$$

For brevity we write t^* for $t_z[(\lambda\vec{x}x.p)\vec{q}q]$ and t' for $t_z[(\lambda\vec{x}.p_x[q])\vec{q}]$. The proof is by induction on $\#r^*$.

$$\begin{aligned} \#((\lambda xr)s\vec{t})^* &= (\#(r_x[s]\vec{t})^*) + 1 \\ &> (\#(r_x[s]\vec{t})') + 1 \\ &= \#((\lambda xr)s\vec{t})', \end{aligned}$$

where the $>$ follows by the induction hypothesis. Note that for the application of the induction hypothesis here we have used $x \in FVr$, which follows from our assumption that we are dealing with λ-I-terms.

$$\begin{aligned} \#(y\vec{t})^* &= \sum_i \#t_i^*\vec{y}_i \\ &> \sum_i \#t_i'\vec{y}_i \\ &= \#(y\vec{t})' \\ \#(z\vec{t})^* &= \#(\lambda\vec{x}x.p)\vec{q}q\vec{t^*} \\ &= (\#(\lambda\vec{x}.p_x[q])\vec{q}\vec{t^*}) + 1 \\ &\geq (\#(\lambda\vec{x}.p_x[q])\vec{q}\vec{t'}) + 1 \\ &> \#(\lambda\vec{x}.p_x[q])\vec{q}\vec{t'} \\ &= \#(z\vec{t})'. \quad \square \end{aligned}$$

Furthermore, it is easy to estimate $\#r$ in terms of $|r|_0$:

Lemma 1.6.4 *Let m be the maximal number of premisses in formulas taken as free assumptions in a derivation r (of a formula of level 0). Then*

$$\#r \leq m^{|r|_0}.$$

The proof is by induction on the generation of $|r|_0 \leq a$.

$$\begin{aligned}
\#(\lambda x r)s\vec{t} &= (\#r_x[s]\vec{t}) + 1 \\
&\leq m^{|r_x[s]\vec{t}|_0} + 1 \\
&\leq m^{|(\lambda x r)s\vec{t}|_0} \\
\#x\vec{t} &= \sum_{i=1}^{n} \#t_i\vec{y}_i \\
&\leq \sum_{i=1}^{n} m^{|t_i\vec{y}_i|_0} \\
&\leq n \cdot m^{\max(|t_1\vec{y}_1|_0,\dots,|t_n\vec{y}_n|_0)} \\
&\leq m^{|x\vec{t}|_0} \qquad \text{since } n \leq m. \qquad \Box
\end{aligned}$$

We now give an estimate for the height of some k-tree for r, for a sufficiently big k. Here we need the notion of the height of a term r, which is defined by

1. height $x = 0$
2. height $\lambda x r =$ (height r) $+ 1$
3. height $ts = \max(($height $t) + 1, ($height $s) + 1)$

Lemma 1.6.5 *For any variable x we have, with an arbitrary k,*

$$|x\vec{y}|_k \leq \textit{level } x. \qquad (1.3)$$

For any term r, if all subterms of r have levels $\leq k$, then

$$|r\vec{y}|_k \leq k + (\textit{height } r). \qquad (1.4)$$

Proof:1.3 can be seen easily by induction on the level of x:

$$|y_i\vec{z}_i|_k \leq \text{ level } y_i < \text{ level } x$$

for $i = 1, \dots, n$ by induction hypothesis, and hence

$$|x\vec{y}|_k \leq \text{ level } x.$$

1.4 is proved by induction on the height of r.

(i) If r is a variable, the claim follows from 1.3

(ii) $(\lambda xr)y\vec{y}$. By induction hypothesis,

$$\begin{aligned} |r_x[y]\vec{y}|_k &\le k + (\text{height } r) \\ &< k + (\text{height } \lambda xr). \end{aligned}$$

Hence, by rule 1,

$$|(\lambda xr)y\vec{y}|_k \le k + (\text{height } \lambda xr).$$

(iii) $rt\vec{y}$. By induction hypothesis,

$$\begin{aligned} |ry\vec{y}|_k &\le k + \text{height } r < k + \text{height } rt \\ |t\vec{z}|_k &\le k + \text{height } t < k + \text{height } rt \end{aligned}$$

and by 1.3

$$|y_i\vec{z}_i|_k \le \text{ level } y_i < k.$$

Hence, by rule 3,

$$|rt\vec{y}|_k \le k + \text{height } rt. \qquad \square$$

In Lemma 1.6.3 we needed the hypothesis that r is a λ-I-term in order to conclude $\#r > \#r'$ from $r \xrightarrow{1} r'$. This hypothesis is certainly necessary, since in non-λ-I-terms subterms can disappear by means of conversions, and hence the head reduction tree may not show any trace of a conversion inside the term. An example is $(\lambda xy)((\lambda xp)q)$ and $(\lambda xy)(p_x[q])$, both of which have the same head reduction tree (consisting of one additional node labeled y).

However, we can easily reduce the general case to the case of λ-I-terms. To achieve this we just introduce dummy variables which turn the given term r into a λ-I-term r^* (a variant of r, as we shall say), and note that the length of any reduction sequence for r is bounded by the length of a reduction sequence for r^*.

By an *immediate variant* of a term r of type $\vec{A} \to P$ we mean a term

$$r' \equiv \lambda\vec{y}.ut(r\vec{y}),$$

where t is any term of some type B and u is a new variable of type $B, P \to P$: the variables $\vec{y}$ are supposed to have types $\vec{A}$. Note that r' has the same type $\vec{A} \to P$ as r. Call a term $r^{(m)}$ an *m-fold immediate variant* of r if there are terms $r^{(0)}, r^{(1)}, \dots, r^{(m-1)}$ such that $r^{(o)} \equiv r$ and $r^{(i+1)}$ is an immediate variant of $r^{(i)}$. Finally a term r^* is called a *variant* of r if it is obtained from r by taking possibly multiple immediate variants of all of its subterms. More precisely, r^* is a variant of r if variant rr^* can be derived by the rules

1. variant $xx^{(m)}$
2. If variant rr^*, then variant $(\lambda xr)(\lambda xr^*)^{(m)}$.
3. If variant tt^* and variant ss^*, then variant $(ts)(t^*s^*)^{(m)}$.

It is obvious that for any term we can find a variant which is a λ-I-term: just replace any subterm λxr with $x \notin FVr$ by its immediate variant

$$\lambda x\vec{y}.ux(r\vec{y}).$$

As noted above, it suffices to prove

Lemma 1.6.6 *If $r \xrightarrow{1} r_1$ and r^* is a variant of r, then we can find a variant r_1^* of r_1 such that $r^* \rightarrow^+ r_1^*$, where $\rightarrow^+$ is defined just as $\rightarrow^*$ exept that reflexivity is not allowed.*

For the proof we need two observations.

$$r's \text{ converts into } (rs)' \tag{1.5}$$

$$(\lambda\vec{x}.(\lambda xr)^{(m)})\vec{s}s \rightarrow^* (\lambda\vec{x}.((\lambda xr)s)^{(m)})\vec{s} \tag{1.6}$$

1.5 follows from the fact that

$$(\lambda y\vec{y}.ut(ry\vec{y}))s \text{ converts into } \lambda\vec{y}.ut(rs\vec{y}).$$

1.6 is proved by induction on m. The case $m = 0$ is immediate. Suppose $m > 0$. Writing p for $(\lambda xr)^{(m-1)}$, we have

$$\begin{aligned}
(\lambda\vec{x}.(\lambda xr)^{(m)}\vec{s}s &\equiv (\lambda\vec{x}.p')\vec{s}s \\
&\equiv (\lambda\vec{x}.\lambda y\vec{y}.ut(py\vec{y}))\vec{s}s \\
&\xrightarrow{1} (\lambda\vec{x}.\lambda\vec{y}.ut(ps\vec{y}))\vec{s} \\
&\equiv (\lambda\vec{x}.(ps)')\vec{s} \\
&\equiv (\lambda\vec{x}.((\lambda xr)^{(m-1)}s)')\vec{s} \\
&\rightarrow^* (\lambda\vec{x}.((\lambda xr)s)^{(m)})\vec{s},
\end{aligned}$$

where in the last step we have used 1.5.

For the proof of the Lemma we restrict ourselves to the case

$$(\lambda x_1 x.r)s_1 s \xrightarrow{1} (\lambda x_1 r_x[s])s_1;$$

the other cases are similar or immediate by the induction hypothesis. An arbitrary variant of $(\lambda x_1 x.r)s_1 s$ must have the form

$$(((\lambda x_1(\lambda x r^*)^{(m)})^{(m_1)}s_1^*)^{(n_1)}s^*)^{(n)}$$

Because of 1.3 we may assume that this variant has the more special form

$$(\lambda x_1(\lambda x r^*)^{(m)})s_1^* s^*.$$

Because of 1.4 we know that this term reduces to

$$(\lambda x_1((\lambda x r^*)s^*)^{(m)})s_1^*.$$

Now this term clearly reduces to

$$(\lambda x_1(r_x^*[s^*])^{(m)})s_1^*,$$

which is a variant of $(\lambda x_1 r_x[s])s_1$. □

To summarize, we have the following result.

Theorem 1.6.1 *Let r be a derivation of a formula of level 0, i. e. of a propositional variable. Let r^* be a λ-I-variant of r. Let k and m be such that all subterms of r have levels $\leq k$, and that all assumptions free in r have $\leq m$ premisses. Then the length of an arbitrary reduction sequence for r with respect to $\xrightarrow{1}$ is bounded by*

$$m^{2_k(k+(\text{height } r^*))}. \quad \square$$

Chapter 2

Normalization for first–order logic

We restrict our attention to the $\to\forall$–fragment of first–order logic with just introduction and elimination rules for both symbols, *i. e.* with minimal logic formulated in natural deduction style. This restriction does not mean a loss in generality, since it is well known that full classical first–order can be embedded in this system; the argument for that fact is sketched in section 2.1. Equality is not treated as a logical symbol, but can be added via suitable equality axioms.

We extend our results and estimates on normalization to first–order logic by the method of collpasing types. Applications include the subformula property, Herbrand's theorem and the interpolation theorem.

2.1 The $\to\forall$–fragment of first–order logic as a typed λ–calculus

Assume that a fixed (at most countable) supply of function variables $f, g, h, \dots$ and predicate variables $P, Q, \dots$ is given, each with an arity ≥ 0. *Terms* are built up from object variables x, y, z by means of $fr_1 \dots r_m$. *Formulas* are built up from prime formulas $Pr_1 \dots r_m$ by means of $(A \to B)$ and $\forall xA$. *Derivations* are built up from assumption variables x^A, y^A by means of the rule $\to^+$ of implication introduction

$$(\lambda x^A r^B)^{A \to B},$$

the rule $\to^-$ of implication elimination

$$(t^{A\to B}s^A)^B,$$

the rule $\forall^+$ of $\forall$–introduction

$$(\lambda x r^A)^{\forall x A},$$

provided that no assumption variable y^B free in r^A has x free in its type B, and finally the rule $\forall^-$ of $\forall$–elimination

$$(t^{\forall x A}s)^{A_x[s]}.$$

Each of the rules $\to^+, \forall^+$ and $\forall^-$ has a uniquely determined derivation as their *premiss*, whereas $\to^-$ has the two derivations $t^{A\to B}$ and s^A as premisses. Here $t^{A\to B}$ is called the *main premiss* and s^A is called the *side premiss*.

As an example we give a derivation of

$$\forall x(Px \to Qx) \to (\forall x Px \to \forall x Qx).$$

Such a derivation is

$$\lambda u^{\forall x(Px\to Qx)}\lambda v^{\forall x Px}\lambda x((ux)(vx)).$$

Derivation can be easily written in the more usual tree form. We will continue to use the word term for derivations (as long as this does not lead to confusion with the notion of (object) term inherent in first–order logic), and type for formula.

Note that our ($\to\forall$–fragment of) minimal logic contains full classical first–order logic. This can be seen as follows:

1. Choose a particular propositional variable and denote it by $\bot$ (falsity). Associate with any formula A in the language of classical first–order logic a finite list $\vec{A}$ of formulas in our $\to\forall$–fragment, by induction on A:

$$\begin{array}{rcl}
P\vec{r} & \mapsto & P\vec{r} \\
\neg A & \mapsto & \vec{A} \to \bot \\
A \to B & \mapsto & \vec{A} \to B_1, \dots, \vec{A} \to B_n \\
A \wedge B & \mapsto & \vec{A}, \vec{B} \\
A \vee B & \mapsto & (\vec{A} \to \bot), (\vec{B} \to \bot) \to \bot \\
\forall x A & \mapsto & \forall x A_1, \dots, \forall x A_m \\
\exists x A & \mapsto & \forall x(\vec{A} \to \bot) \to \bot
\end{array}$$

2. In any model $\mathcal{M}$ where $\bot$ is interpreted by falsity, we clearly have that a formula A in the language of full first-order logic holds under an assignment α iff all formulas in the assigned sequence $\vec{A}$ hold under α.

3. Our derivation calculus for the $\to\forall$-fragment is complete in the following sense: A formula A is derivable from stability assumptions

$$\forall\vec{x}(\neg\neg P\vec{x} \to P\vec{x})$$

for all predicate symbols P in A iff A is valid in any model under any assignment.

2.2 Strong normalization

Here we use the method of collapsing types (cf. [TvD88, p.560]) to transfer our results and estimates concerning strong normalization from implicational logic to first-order logic.

The notions concerning conversion introduced in section 1.2 can be easily extended to first-order logic. In particular, we have

$$(\lambda\vec{x}x.r)\vec{s}s \text{ converts into } (\lambda\vec{x}.r_x[s])\vec{s},$$

where the variables $\vec{x}, x$ now can be either assumption variables or else object variables. The rules generating the relation $r \to r'$ are extended by requiring $r \to r$ for object terms r of our first-order logic. Again a derivation is said to be in *normal form* if it does not contain a convertible subderivation.

For any formula A of first-order logic we define its *collapse* A^c by

$$\begin{aligned} (P\vec{r})^c &\equiv P \\ (A \to B)^c &\equiv A^c \to B^c \\ (\forall xA)^c &\equiv \top \to A^c \end{aligned}$$

where $\top :\equiv \bot \to \bot$ with $\bot$ a fixed propositional variable (i. e. $\top$ means truth). The *level* of a formula A of first-order logic is defined to be the level of its collapse A^c. For any derivation r^B in first-order logic we can now define its *collapse* $(r^B)^c$. It is plain from this definition that for any derivation r^B in first-order logic with free assumption variables $x_1^{A_1}, \dots, x_m^{A_m}$ the collapse $(r^B)^c$ is a derivation $(r^c)^{B^c}$ in implicational logic with free assumption variables $x_1^{A_1^c}, \dots, x_m^{A_m^c}$.

$$(x^A)^c \equiv x^{A^c}$$

$$\begin{aligned}(\lambda x^A r)^c &\equiv \lambda x^{A^c} r^c\\ (t^{A\to B} s)^c &\equiv t^c s^c\\ (\lambda x r)^c &\equiv \lambda x^\top r^c\\ (t^{\forall x A} s)^c &\equiv t^c(\lambda z^\perp z^\perp)^\top\end{aligned}$$

Note that for any derivation r^B, assumption variable x^A and derivation s^A we have that $r^c[s^c]$ is a derivation in implicational logic (where the substitution of s^c is done for the assumption variable x^{A^c}), which is the collapse of $r[s]$. Also for any derivation r^B, object variable x and term s we have that $r_x[s]$ is a derivation of $B_x[s]$ with collapse $(r_x[s])^c \equiv r^c$.

Lemma 2.2.1 *If $r \xrightarrow{1} r'$ in first-order logic, then $r^c \xrightarrow{1} (r')^c$ in implicational logic.*

The proof is by induction on the generation of $r \xrightarrow{1} r'$. We only treat the case
$$(\lambda x r)s \xrightarrow{1} r_x[s].$$
If x is an assumption variable, then
$$\begin{aligned}((\lambda x^A r^B)s^A)^c &\equiv (\lambda x^{A^c} r^c)s^c\\ &\xrightarrow{1} r^c[s^c]\\ &\equiv (r[s])^c),\end{aligned}$$
by the note above. If x is an object variable, then
$$\begin{aligned}((\lambda x^A)s)^c &\equiv (\lambda x^\top r^c)(\lambda z^\perp z^\perp)^\top\\ &\xrightarrow{1} r^c\\ &\equiv (r[s])^c),\end{aligned}$$
again by the note above. □

Hence from Theorem 1.5.1 we can conclude

Theorem 2.2.1 *Any derivation r in first-order logic is strongly normalizable.* □

Also we can apply Theorem 1.6.1 to obtain an upper bound for the length of arbitrary reduction sequences.

Theorem 2.2.2 *Let r be a derivation in first-order logic of a formula of level 0, i. e. a prime formula. Let r^c be the collapse of r into implicational logic, and r^{c*} be a λ-I-variant of r^c (cf. the remark before Lemma 1.6.6). Let k and m be such that all subderivations of r^{c*} have levels $\leq k$, and all assumption variables free in r^{c*} have $\leq m$ premisses. Then the length of an arbitrary reduction sequence for r with respect to $\xrightarrow{1}$ is bounded by*

$$m^{2_k(k+\text{height } r^{c*})}. \quad \Box$$

2.3 Uniqueness

The Church–Rosser Theorem and hence the uniqueness of the normal form for derivations in first-order logic can be proved exactly as in section 1.3. We do not repeat this here.

2.4 Applications

Here we want to draw some conclusions from the fact that any derivation in first-order logic can be transformed into normal form. The arguments in this section are based on Prawitz' book [Pra65]. We begin with an analysis of the form of normal derivations.

Let a derivation r be given. A sequence $r_1, \ldots, r_m$ of subderivations of r is a *path* if

1. r_1 is an assumption variable,
2. r_i is the main premiss of r_{i+1}, and
3. r_m is either the whole derivation r or else the side premiss of an instance of the rule $\rightarrow^-$ within r.

It is obvious that any subderivation of r belongs to exactly one path. The *order* of the path ending with the whole derivation r is defined to be 0, and if the order of the path through the main premiss t of some instance $t^{A\rightarrow B}s^A$ of the rule $\rightarrow^-$ in r is k, then the order of the path ending with that s^A is defined to be $k+1$.

The relation "A is a *subformula* of B" is defined to be the transitive and reflexive closure of the relation "immediate subformula", defined by

1. A and B are immediate subformulas of $A \rightarrow B$,

2. $A_x[r]$ is an immediate subformula of $\forall x A$.

We will also need the notion "A is a *strictly positive subformula* of B", which is defined to be the transitive and reflexive closure of the relation "immediate strictly positive subformula", defined by

1. B is an immediate strictly positive subformula of $A \to B$,

2. $A_x[r]$ is an immediate strictly positive subformula of $\forall x A$.

In a normal derivation r any path $r_1^{A_1}, \ldots, r_m^{A_m}$ has a rather perspicious form: all elimination rules must come before all introduction rules. Hence, if i is maximal such that $r_i^{A_i}$ ends with an elimination rule, then A_i must be a strictly positive subformula of all A_j for $j \neq i$. This A_i is called the *minimal formula* of the path. Also, any A_j with $j \leq i$ is a strictly positive subformula of A_1, and any A_j with $j \geq i$ is a strictly positive subformula of A_m.

Theorem 2.4.1 (Subformula property) *If r^A is a normal derivation with free assumption variables among $x_1^{A_1}, \ldots, x_m^{A_m}$ and s^B is a subderivation of r^A, then B is a subformula of A or of some A_i.*

The proof is by induction on the order of paths in r, using the property of paths in normal derivations mentioned above. □

We write $A_1, \ldots, A_m \vdash A$ to mean that there is a derivation r^A with free assumption variables among $x_1^{A_1}, \ldots, x_m^{A_m}$.

Theorem 2.4.2 (Herbrand) *Assume $\forall \vec{x}_1 A_1, \ldots, \forall \vec{x}_m A_m \vdash B$ with quantifier-free $A_1, \ldots, A_m, B$. Then we can find $\vec{r}_{11}, \ldots, \vec{r}_{1n_1}, \ldots, \vec{r}_{m1}, \ldots, \vec{r}_{mn_m}$ such that*

$$A_1[\vec{r}_{11}], \ldots, A_1[\vec{r}_{1n_1}], \ldots, A_m[\vec{r}_{m1}], \ldots, A_m[\vec{r}_{mn_m}] \vdash B$$

Proof: To simplify notation let us assume $\forall x A \vdash B$ with quantifier-free A, B. By section 2.2 we can construct from the given derivation a normal derivation r^B with free assumption variables among $x^{\forall x A}$. By induction on the order of paths it is easy to see that any path must end with the derivation of a quantifier-free formula and must begin with the rule $\forall^-$, i. e. with $x^{\forall x A} r_i$. Now replace any such subderivation by $y_i^{A[r_i]}$, with new assumption variables y_i. □

Our next application is the Craig interpolation theorem. We shall use the notation $A_1, \ldots, A_m \vdash^c A$ (c for classical) to mean that there is a derivation r^A with free assumption variables among $x_1^{A_1}, \ldots, x_m^{A_m}$ and some stability assumptions $y^{\forall \vec{x}(\neg\neg P\vec{x} \to P\vec{x})}$ for P predicate variable in $\vec{A}, A$, where again $\neg B$ denotes $B \to \bot$ with a fixed propositional variable $\bot$.

Theorem 2.4.3 (Interpolation) *Assume $\Gamma, \Delta \vdash A$. Then we can find a finite list $\vec{F}$ of formulas such that*

$$\Gamma \vdash^c \vec{F} \text{ and } \vec{F}, \Delta \vdash^c A$$

(where $\Gamma \vdash^c \vec{F}$ means $\Gamma \vdash^c F_i$ for each F_i in $\vec{F}$), and any object or predicate variable free in $\vec{F}$ occurs free both in Γ and in Δ, A.

For the proof we shall use a somewhat more explicit formulation of the theorem: Let r^A be a derivation with free assumption variables among $\vec{u}^\Gamma, \vec{v}^\Delta$. Then we can find a finite list $\vec{r_1}^{\vec{F}}$ of derivations with free assumption variables among $\vec{u}^\Gamma$ and stability assumptions and a derivation r_2^A with free assumption variables among $\vec{y}^{\vec{F}}, \vec{v}^\Delta$ and stability assumptions, such that any object or predicate variable free in $\vec{F}$ occurs free both in Γ and in Δ, A.

For brevity we shall not mention stability assumptions any more (they will only be used in case 2*b*(*ii*) below), and write "r^A with $\vec{u}^\Gamma$", to mean the derivation r^A with free assumption variables among $\vec{u}^\Gamma$.

The proof is by induction on the height of the given derivation, which by section 2.2 we can assume to be normal. We distinguish two cases according to whether it ends with an introduction rule (i. e. $\to^+$ or $\forall^+$) or with an elimination rule.

Case 1a. $(\lambda x^A r^B)^{A \to B}$ with $\vec{u}^\Gamma, \vec{v}^\Delta$. By induction hypothesis for r^B with $x^A, \vec{u}^\Gamma, \vec{v}^\Delta$ we have $\vec{r}_1 \vec{F}$ with $\vec{u}^\Gamma$ and r_2^B with $\vec{y}^{\vec{F}}, x^A, \vec{v}^\Delta$. An application of $\to^+$ to the latter derivation yields $(\lambda x^A r_2^B)^{A \to B}$ with $\vec{y}^{\vec{F}}, \vec{v}^\Delta$.

Case 1b. $(\lambda x r^A)^{\forall x A}$ with $\vec{u}^\Gamma, \vec{v}^\Delta$, where x is not free in Γ, Δ. By induction hypothesis for r^A with $\vec{u}^\Gamma, \vec{v}^\Delta$ we have $\vec{r}_1^{\vec{F}}$ with $\vec{u}^\Gamma$ and r_2^A with $\vec{y}^{\vec{F}}, \vec{v}^\Delta$. Since x is not free in Γ, we know that x is not free in $\vec{F}$. An application of $\forall^+$ to the latter derivation yields $(\lambda x r_2^A)^{\forall x A}$ with $\vec{y}^{\vec{F}}, \vec{v}^\Delta$.

Case 2a. $(w^{C \to D} s^C \vec{t})^A$ with $\vec{u}^\Gamma, \vec{v}^\Delta$.

Subcase i. $w^{C \to D}$ is among $\vec{u}^\Gamma$. By induction hypothesis for s^C with $\vec{u}^\Gamma, \vec{v}^\Delta$ we have $\vec{s}_1 \vec{F}$ with $\vec{v}^\Delta$ and s_2^C with $\vec{y}^{\vec{F}}, \vec{u}^\Gamma$. By induction hypothesis for $(u^D \vec{t})^A$ with $u^D, \vec{u}^\Gamma, \vec{v}^\Delta$ we have $\vec{t}^{\vec{G}}$ with $u^D, \vec{u}^\Gamma$ and t_2^A with $\vec{z}^{\vec{G}}, \vec{v}^\Delta$. From these derivations we obtain

$$(\lambda \vec{y}^{\vec{F}} (\vec{t_1}^{\vec{G}})_u [w^{C \to D} s_2^C])^{\vec{F} \to \vec{G}} \text{ with } \vec{u}^\Gamma$$

and

$$(t_2^A)_{\vec{z}} [\vec{x}^{\vec{F} \to \vec{G}} \vec{s}_1^{\vec{F}}] \text{ with } \vec{x}^{\vec{F} \to \vec{G}}, \vec{v}^\Delta,$$

where $\vec{F} \to \vec{G}$ means $\vec{F} \to G_1, \ldots, \vec{F} \to G_n$.

Subcase ii. $w^{C \to D}$ is among $\vec{v}^{\Delta}$. By induction hypothesis for s^{C} with $\vec{u}^{\Gamma}, \vec{v}^{\Delta}$ we have $\vec{s}_1^{\vec{F}}$ with $\vec{u}^{\Gamma}$ and s_2^{C} with $\vec{y}^{\vec{F}}, \vec{v}^{Delta}$. By induction hypothesis for $(u^{D}\vec{t})^{A}$ with $u^{D}, \vec{u}^{\Gamma}, \vec{v}^{\Delta}$ we have $\vec{t}_1^{\vec{G}}$ with $\vec{u}^{\Gamma}$ and t_2^{A} with $\vec{z}^{\vec{\Gamma}}, u^{D}, \vec{v}^{\Delta}$. From these derivations we obtain

$$(\vec{s}_1, \vec{t}_1)^{\vec{F}, \vec{G}} \text{ with } \vec{u}^{\Gamma}$$

and

$$(t_2^{A})_u[w^{C \to D} s_2^{C}] \text{ with } \vec{y}^{\vec{F}}, \vec{z}^{\vec{G}}, \vec{v}^{\Delta}.$$

Case 2b. $w^{\forall x C} s \vec{t}$ with $\vec{u}^{\Gamma}, \vec{v}^{\Delta}$.

Subcase i. $w^{\forall x C}$ is among $\vec{u}^{\Gamma}$. By induction hypothesis for $(u^{C[s]}\vec{t})^{A}$ with $u^{C[s]}, \vec{u}^{\Gamma}, \vec{v}^{\Delta}$ we have $\vec{t}_1^{\vec{F}}$ with $u^{C[s]}, \vec{u}^{\Gamma}$ and t_2^{A} with $\vec{y}^{\vec{F}}, \vec{v}^{\Delta}$. Let $\vec{z}$ be all variables free in $\vec{F}$ that are in s, but not free in Γ. We now construct derivations

$$(\lambda \vec{z}(\vec{t}_1^{\vec{F}})_u[w^{\forall x C} s])^{\forall \vec{z} \vec{F}} \text{ with } \vec{u}^{\Gamma}$$

and

$$(t_2^{A})_{\vec{y}}[\vec{z}^{\forall \vec{z} \vec{F}} \vec{z}] \text{ with } \vec{x}^{\forall \vec{z} \vec{F}}, \vec{v}^{\Delta},$$

where $\forall \vec{z} \vec{F}$ means $\forall \vec{z} F_1, \ldots, \forall \vec{z} F_m$. Note that any object or predicate variable free in $\forall \vec{z} \vec{F}$ is both free in Δ, A and free in Γ.

Subcase ii. $w^{\forall x C}$ *is among* $\vec{v}^{\Delta}$. By induction hypothesis for $(u^{C[s]}\vec{t})^{A}$ with $u^{C[s]}, \vec{u}^{\Gamma}, \vec{v}^{\Delta}$ we have $\vec{t}_1^{\vec{G}}$ with $\vec{u}^{\Gamma}$ and t_2^{A} with $\vec{y}^{\vec{G}}, u^{C[s]}, \vec{v}^{\Delta}$. Let $\vec{z}$ be all variables free in $\vec{G}$ that are in s, but not free in Δ, A. We now construct derivations

$$(\lambda v^{\forall \vec{z} \neg \vec{G}}(v \vec{z} \vec{t}_1))^{\neg \forall \vec{z} \neg \vec{G}} \text{ with } \vec{u}^{\Gamma}$$

and

$$(t^{\neg\neg A \to A} \lambda v^{\neg A}. x^{\neg \forall \vec{z} \neg \vec{G}} \lambda \vec{z} \lambda \vec{y}(v(t_2^{A})_u[w^{\forall x C} s]))^{A} \text{ with } x^{\neg \forall \vec{z} \neg \vec{G}}, \vec{v}^{\Delta}$$

and stability assumptions (which are used to build $t^{\neg\neg A \to A}$). Note again that any object or predicate variable free in $\neg \forall \vec{z} \neg \vec{G}$ is both free in Γ and free in Δ, A. □

Chapter 3

Normalization for arithmetic

3.1 Ordinal notations

We want to discuss the derivability and underivability of initial cases of transfinite induction in arithmetical systems. In order to do that we shall need some knowledge and notations for ordinals. Now we do not want to assume set theory here; hence we introduce a certain initial segment of the ordinal (the ordinals $< \varepsilon_0$) in a formal, combinatorial way, i. e. via ordinal notations. Our treatment is based on the Cantor normal form for ordinals (cf. Bachmann [Bac55]). We also introduce some elementary relations and operations for such ordinal notations, which will be used later.

Definition 3.1.1 *We define the two notions*

- *α is an ordinal notation*
- *$\alpha < \beta$ for ordinal notations α, β*

simultaneously by induction:

1. *If $\alpha_m, \ldots, \alpha_0$ are ordinal notations and $\alpha_m \geq \ldots \geq \alpha_0$ (where $\alpha \geq \beta$ means $\alpha > \beta$ or $\alpha = \beta$), then*

$$\omega^{\alpha_m} + \cdots + \omega^{\alpha_0}$$

is an ordinal notation. Note that the empty sum denoted by 0 is allowed here.

2. *If $\omega^{\alpha_m} + \cdots + \omega^{\alpha_0}$ and $\omega^{\beta_n} + \cdots + \omega^{\beta_0}$ are ordinal notations, then*

$$\omega^{\alpha_m} + \cdots + \omega^{\alpha_0} < \omega^{\beta_n} + \cdots + \omega^{\beta_0}$$

iff there is an $i \geq 0$ such that $\alpha_{m-i} < \beta_{n-i}$, $\alpha_{m-i+1} = \beta_{n-i+1}, \ldots, \alpha_m = \beta_n$, or else $m < n$ and $\alpha_m = \beta_n, \ldots, \alpha_0 = \beta_{n-m}$

It is easy to see (by induction on the levels in the inductive definition) that $<$ is a linear order with 0 being the smallest element.

We shall use the notation 1 for ω^0, a for $\omega^0 + \cdots + \omega^0$ with a copies of ω^0 and $\omega^\alpha a$ for $\omega^\alpha + \cdots + \omega^\alpha$ again with a copies of ω^α.

Definition 3.1.2 *of addition for ordinal notations:*

$$\omega^{\alpha_m} + \cdots + \omega^{\alpha_0} + \omega^{\beta_n} + \cdots + \omega^{\beta_0} := \omega^{\alpha_m} + \cdots + \omega^{\alpha_i} + \omega^{\beta_n} + \cdots + \omega^{\beta_0}$$

where i is minimal such that $\alpha_i \geq \beta_n$.

It is easy to see that $+$ is an associative operation which is strictly monotonic in the second argument and weakly monotonic in the first argument. Note that $+$ is not commutative: $1 + \omega = \omega \neq \omega + 1$.

Definition 3.1.3 *of natural (or Hessenberg) addition of ordinal notations:*

$$(\omega_{\alpha_m} + \cdots + \omega^{\alpha_0})\#(\omega_{\beta_n} + \cdots + \omega^{\beta_0}) := \omega^{\gamma_{m+n}} + \cdots + \omega^{\gamma_0}$$

where $\gamma_{m+n}, \ldots, \gamma_0$ is a decreasing permutation of $\alpha_m, \ldots, \alpha_0, \beta_n, \ldots, \beta_0$.

Again it is easy to see that # is associative, commutative and strictly monotonic in both arguments.

We will also need to know how ordinal notations of the form $\beta + \omega^\alpha$ can be approximated from below. First note that

$$\delta < \alpha \rightarrow \beta + \omega^\delta a < \beta + \omega^\alpha.$$

Furthermore, for any $\gamma < \beta + \omega^\alpha$ we can find a $\delta < \alpha$ and an a such that

$$\gamma < \beta + \omega^\delta a.$$

Definition 3.1.4 *of 2^α for ordinal notations α. Let $\alpha_m \geq \cdots \alpha_0 \geq \omega > k_n \geq \cdots \geq k_1 > 0$. Then*

$$2^{\omega^{\alpha_m} + \cdots + \omega^{\alpha_0} + \omega^{k_n} + \cdots + \omega^{k_1} + \omega^0 a} := \omega^{\omega^{\alpha_m} + \cdots + \omega^{\alpha_0} + \omega^{k_n - 1} + \cdots + \omega^{k_1 - 1}} 2^a.$$

It is easy to see that $2^{\alpha+1} = 2^\alpha + 2^\alpha$ and that 2^α is strictly monotonic in α.

In order to work with ordinal notations in a purely arithmetical system we set up a bijection between ordinal notations and nonnegative integers (i. e., a Gödel numbering). For its definition it is useful to refer to ordinal notations in the form

$$\omega^{\alpha_m} a_m + \cdots + \omega^{\alpha_0} a_0 \text{ with } \alpha_m > \cdots > \alpha_0.$$

Definition 3.1.5 *For any ordinal notation α we define its Gödel number $|\alpha|$ inductively by*

$$|0| := 0,$$
$$|\omega^{\alpha_m} a_m + \cdots + \omega^{\alpha_0} a_0| := (\prod_{i \leq m} p_{|\alpha_i|}^{a_i}) - 1.$$

For any nonnegative integer x we define its corresponding ordinal notation ox inductively by

$$o0 = 0$$
$$o((\prod_{i \leq m} p_i^{a_i}) - 1) = \sum_{i \leq m} \omega^{oi} a_i$$

where the sum is to be understood as the natural sum.

Lemma 3.1.1 *1. $o|\alpha| = \alpha$,*

2. $|ox| = x$. □

This can be proved easily by induction.

Hence we have a bijection between ordinal notations and nonnegative integers. Using this bijection we can transfer our relations and operations on ordinal notations to computable relations and operations on nonnegative integers. We will use the notations

$x \prec y$	for	$ox < oy$
ω^x	for	$\lvert\omega^{ox}\rvert$
$x \oplus y$	for	$\lvert(ox) + (oy)\rvert$.

3.2 Provable initial cases of transfinite induction

We now set up some formal systems of arithmetic and derive initial cases of the principle of transfinite induction in them, i. e. of

$$\forall x(\forall y \prec x : Py \rightarrow Px) \rightarrow \forall x \prec a : Px$$

for some numeral a and a predicate variable P. In section 3.4 we will see that our results here are optimal in the sense that for larger segments of the ordinals transfinite induction is underivable. All these results are due to Gentzen [Gen43].

Our arithmetical systems are based on a fixed (possibly countably infinite) supply of function constants and predicate constants which are assumed to denote fixed functions and predicates on the nonnegative integers for which a computation procedure is known. Among the function constants there must be a constant S for the successor function and 0 for (the 0-place function) zero. Among the predicate constants there must be a constant $=$ for equality and $\bot$ for (the 0-place predicate) falsity. In order to formulate the general principle of transfinite induction we also assume that predicate variables $P, Q, \dots$ are present.

Terms are built up from object variables x, y, z by means of $fr_1 \dots r_m$, where f is a function constant. We identify closed terms which have the same value; this is a convenient way to express in our formal systems the assumption that for each function constant a computation procedure is known. Terms of the form $SS \dots S0$ are called ***numerals.*** We use the notation S^i0 or even i for them. ***Formulas*** are built up from prime formulas $Pr_1 \dots r_m$ with P a predicate constant or a predicate variable by means of $(A \rightarrow B)$ and $\forall x A$. As usual we abbreviate $A \rightarrow \bot$ by $\neg A$.

The ***axioms*** of our arithmetical systems will always include the Peano–axioms

$$\forall xy(Sx = Sy \rightarrow x = y),$$

$$\forall x(Sx \neq 0).$$

Any instance of the induction scheme

$$A[0], \forall x(A[x] \rightarrow A[Sx]) \rightarrow \forall x A[x]$$

with A an arbitrary formula is an axiom of full arithmetic Z. We will also consider subsystems Z_k of Z where the formulas A in the induction scheme are restricted to Π^0_k-formulas; the latter notion is defined inductively, as follows.

1. Any prime formula $P\vec{r}$ is a Π^0_k-formula, for any $k \geq 1$.
2. If A is quantifier-free and B is a Π^0_k-formula, then $A \to B$ is a Π^0_k-formula.
3. If A is a Π^0_k-formula and B is a Π^0_l-formula, then $A \to B$ is a Π^0_p-formula with $p = \max(k+1, l)$.
4. If A is a Π^0_k-formula, then so is $\forall x A$.

Note that a formula is a Π^0_k-formula iff it is logically equivalent[1] to a formula with a prefix of k alternating quantifiers beginning with $\forall$ and a quantifier free kernel. For example, $\forall x \exists y \forall z Pxyz$ is a Π^0_3-formula. In addition, in any arithmetical system we have the equality axioms

$$\forall x(x = x),$$

$$\forall \vec{x}\vec{y}(x_1 = y_1, \ldots, x_m = y_m \to f\vec{x} = f\vec{y}),$$

$$\forall \vec{x}\vec{y}(x_1 = y_1, \ldots, x_m = y_m, P\vec{x} \to P\vec{y})$$

for any function constant f and predicate constant or predicate variable P. We also require for any such P the stability axioms

$$\forall \vec{x}(\neg\neg P\vec{x} \to P\vec{x}).$$

We express our assumption that for any predicate constant a decision procedure is known by adding the axiom

$$P(S^{i_1}0)\ldots(S^{i_m}0)$$

whenever $P\vec{i}$ is true, and

$$\neg P(S^{i_1}0)\ldots(S^{i_m}0)$$

whenever $P\vec{i}$ is false.

We finally allow in any of our arithmetical systems an arbitrary supply of true Π^0_1-formuals as axioms. Our (positive and negative) results concerning

[1]This means logically equivalent in classical logic, i. e. both implications are derivable by the rules of (minimal logic as given in) chapter 2 together with stability axioms.

initial cases of transfinite recursion will not depend on which of those axioms we have chosen, except that for the positive results we always assume

$$\forall x(x \not\prec 0) \tag{3.1}$$

$$\forall xy(z \prec y \oplus \omega^0, z \not\prec y, z \neq y \rightarrow \bot) \tag{3.2}$$

$$\forall x(x \oplus 0 = x) \tag{3.3}$$

$$\forall xyz(x \oplus (y \oplus z) = (x \oplus y) \oplus z) \tag{3.4}$$

$$\forall x(0 \oplus x = x) \tag{3.5}$$

$$\forall x(\omega^x 0 = 0) \tag{3.6}$$

$$\forall xy(\omega^x(Sy) = \omega^x y \oplus \omega^x) \tag{3.7}$$

$$\forall xyz(z \prec y \oplus \omega^x, x \neq 0, x \rightarrow z \prec y \oplus \omega^{fxyz}(gxyz)) \tag{3.8}$$

$$\forall xyz(z \prec y \oplus \omega^x, x \neq 0 \rightarrow fxyz \prec x) \tag{3.9}$$

where in 3.9 f and g are function constants.

Theorem 3.2.1 (Gentzen) *Transfinite induction up to ω_n (with $\omega_1 := \omega, \omega_{n+1} := \omega^{\omega_n}$) i. e. the formula*

$$\forall x(\forall y \prec x : A[y] \rightarrow A[x]) \rightarrow \forall x \prec \omega_n : A[x]$$

is derivable in Z.

Proof: To any formula A we assign a formula A^+ (with respect to a fixed variable x) by

$$A^+ :\equiv \forall y(\forall z \prec y : A_x[z] \rightarrow \forall z \prec y \oplus \omega^x : A_x[z]).$$

We first show

$$A \text{ is progressive } \rightarrow A^+ \text{ is progressive},$$

where "B is ***progressive***" means $\forall x(\forall y \prec x : B[y] \rightarrow B[x])$. So assume that A is progressive and

$$\forall y \prec x : A^+[y]. \tag{3.10}$$

We have to show $A^+[x]$. So assume further

$$\forall z \prec y : A[z] \tag{3.11}$$

and $z \prec y \oplus \omega^x$. We have to show $A[z]$. Case $x = 0$. From $z \prec y \oplus \omega^0$ we have by 3.2 $z \prec y \vee z = y$. If $z \prec y$, then $A[z]$ follows from 3.11, and if $z = y$, then $A[z]$ follows from 3.11 and the progressiveness of A. Case $x \neq 0$. From $z \prec y \oplus \omega^x$ we obtain $z \prec y \oplus \omega^{fxyz} gxyz$ by 3.8 and $fxyz \prec x$ by 3.9. From 3.10 we obtain $A^+[fxyz]$. By the definition of A^+ we get

$$\forall u \prec y \oplus \omega^{fxyz} v : A[u] \to \forall u \prec (y \oplus \omega^{fxyz} v) \oplus \omega^{fxyz} : A[u]$$

and hence, using 3.4 and 3.7

$$\forall u \prec y \oplus \omega^{fxyz} v : A[u] \to \forall u \prec (y \oplus \omega^{fxyz}(Sv) : A[u].$$

Also from 3.11 and 3.6, 3.3 we obtain

$$\forall u \prec y \oplus \omega^{fxyz} 0 : A[u].$$

Using an appropriate instance of the induction scheme we can conclude

$$\forall u \prec y \oplus \omega^{fxyz} gxyz : A[u]$$

and hence $A[z]$.

We now show, by induction on n, how to obtain a derivation of

$$\forall x (\forall y \prec x : A[y] \to A[x]) \to \forall x \prec \omega_n : A[x].$$

So assume the left-hand side, i. e. assume that A is progressive. Case 0. From $x \prec \omega_0$ we get $x = 0$ by 3.5, 3.2 and 3.1, and $A[0]$ follows from the progressiveness of A by 3.1. Case $n + 1$. Since A is progressive, by what we have shown above also A^+ *is progressive*. Applying the induction hypothesis to A^+ yields $\forall x \prec \omega_n : A^+[x]$, and hence $A^+[\omega_n]$ by the progressiveness of $A^+[x]$. Now the definition of A^+ (together with 3.1 and 3.5) yields $\forall z \prec \omega^{\omega_n} : A[z]$. $\square$

Note that in these derivations the induction scheme was used for formulas of unbounded complexity.

We now want to refine Theorem 3.2.1 to a corresponding result for the subsystems Z_k of Z. Note first that if A is a Π^0_k-formula, then the formula A^+ constructed in the proof of Theorem 3.2.1 is a Π^0_{k+1}-formula, and for the proof of

$$A \text{ is progressive } \to A^+ \text{ is progressive}$$

we have used induction with a Π^0_k induction formula.

Now let A be a Π^0_1-formula, and let $A^0 :\equiv A$, $A^{i+1} :\equiv (A^i)^+$. Then A^k is a Π^0_{k+1}-formula, and hence in Z_k we can derive *that if A is progressive*, then also $A^1, A^2, \ldots A^k$ are all progressive. Let $\omega_1[m] := m, \omega_{i+1}[m] = \omega^{\omega_i[m]}$. Since in

Z_k we can derive that A^k is progressive, we can also derive $A^k[0], A^k[1], A^k[2]$ and generally $A^k[m]$ for any m, i. e. $A^k[\omega_1[m]]$. But since

$$A^m \equiv (A^{k-1})^+ \equiv \forall y(\forall z \prec y : A^{k-1}[z] \rightarrow \forall z \prec y \oplus \omega^x : A^{k-1}[z]),$$

we first get (with $y = 0$) $\forall z \prec \omega_2[m] : A^{k-1}[z]$ and then $A^{k-1}[\omega_2[m]]$ by the progressiveness of A^{k-1}. Repeating this argument we finally obtain $A^0[\omega_{k+1}[m]]$. Hence we have

Theorem 3.2.2 *Let A be a Π^0_1-formula. Then in Z_k we can derive transfinite induction for A up to $\omega_{k+1}[m]$ for any m, i. e.*

$$Z_k \vdash \forall x(\forall y \prec x : A[y] \rightarrow A[x]) \rightarrow \forall x \prec \omega_{k+1}[m] : A[x]. \quad \Box$$

If more generally we start out with a Π^0_l-formula A instead, where $1 \leq l \leq k$, then a similar argument yields

Theorem 3.2.3 *Let A be a Π^0_l-formula, $1 \leq l \leq k$. Then in Z_k we can derive transfinite induction for A up to $\omega_{k+2-l}[m]$ for any m, i. e.*

$$Z_k \vdash \forall x(\forall y \prec x : A[y] \rightarrow A[x]) \rightarrow \forall x < \omega_{k+2-l}[m] : A[x]. \quad \Box$$

Our next aim is to prove that these bounds are sharp. More precisely, we will show that in Z (no matter how many true Π^0_1-formulas we have added as axioms) one cannot derive transfinite induction up to ϵ_0, i. e. the formula

$$\forall x(\forall y \prec x : Py \rightarrow Px) \rightarrow \forall x Px$$

with a free predicate variable P, and that in Z_k one cannot derive transfinite induction up to ω_{k+1}, i. e. the formula

$$\forall x(\forall y \prec x : Py \rightarrow Px) \rightarrow \forall x \prec \omega_{k+1} : Px.$$

This will follow from the method of normalization applied to arithmetical systems, which we have to develop first.

3.3 Normalization for arithmetic with the ω–rule

We will show in section 3.5 that a normalization theorem does not hold for a system of arithmetic like Z in section 3.2, in the sense that for any formula

A derivable in Z there is a derivation of the same formula A in Z which only uses formulas of a level bounded by the level of A. The reason for this failure is the presence of the induction axioms, which can be of arbitrary level.

Here we remove that obstacle against normalization and replace the induction axioms by a rule with infinitely many premises, the so-called ω-rule (suggested by Hilbert and studied by Lorenzen, Novikov and Schütte), which allows to conclude $\forall x A[x]$ from $A[0], A[1], A[2], \ldots$.

Clearly this ω-rule can also be used to replace the rule $\forall^+$. As a consequence we do not need to consider free object variables.

So we introduce the system Z^∞ of ω-arithmetic as follows. Z^∞ has the same language and — apart from the induction axioms — the same axioms as Z. Derivations in Z^∞ are infinite objects; they are built up from assumption variables x^A, y^B and constants ax^A for any axiom A of Z other than an induction axiom by means of the rules

$$(\lambda x^A r^B)^{A \to B}$$

$$(t^{A \to B} s^A)^B$$

$$\langle r_i^{A[i]} \rangle_{i<\omega}^{\forall x A}$$

$$(t^{\forall x A} i)^{A[i]}$$

denoted by $\to^+, \to^-, \omega$ and $\forall^-$, respectively.

More precisely, we define the notion of an $\vec{x}$-derivation[2] (i. e. a derivation in Z^∞ with free assumption variables among $\vec{x}$) of height $\leq \alpha$ and degree $\leq k$ inductively, as follows[3]

* Any assumption variable x^A and any axiom ax^A is an $\vec{x}$-derivation of height $\leq \alpha$ and degree $\leq k$, for any list $\vec{x}$ of assumption variables (*containing x in the first case*), ordinal α and number k.

$\to^+$ If r^B is an $\vec{x}, x, \vec{y}$-derivation of height $\leq \alpha_0 < \alpha$ and degree $\leq k$, then $(\lambda x^A r^B)^{A \to B}$ is an $\vec{x}, \vec{y}$-derivation of height $\leq \alpha$ and degree $\leq k$.

[2]Note that derivations are infinite objects now. They may be viewed as mappings from finite sequences of natural numbers (= nodes in the derivation tree) to lists of data including the formula appearing at that node, the rule applied last, a list of assumption variables including all those free in the subderivation (starting at that node), a bound on the height of the subderivation, and a bound on the degree of the subderivation.

[3]Intuitively, the degree of a derivation is the least number $\geq$ the level of any subderivation $\lambda x r$ in a context $(\lambda x r)s$ or $\langle r_i \rangle_{i<\omega}$ in a context $\langle r_i \rangle_{i<\omega} j$, where the level of a derivation is the level of its type, i.e. the formula it derives. This notion of a degree is needed for the normalization proof we give below.

$\to^-$ If $t^{A\to B}$ and s^A are $\vec{x}$–derivations of heights $\leq \alpha_i < \alpha$ and degrees $\leq k_i \leq k \quad (i = 1, 2)$, then $(t^{A\to B}s^A)^B$ is an $\vec{x}$–derivation of height $\leq \alpha$ and degree $\leq m$ with $m = \max(k, \text{level } (A \to B))$, if $t^{A\to B}$ is generated by the rule $\to^+$, or of degree $\leq k$ otherwise.

ω If $r_i^{A[i]}$ are $\vec{x}$–derivations of heights $\leq \alpha_i < \alpha$ and degrees $\leq k_i \leq k \quad (i < \omega)$, then $\langle r_i^{A[i]} \rangle_{i<\omega}^{\forall x A}$ is an $\vec{x}$–derivation of height $\leq \alpha$ and degree $\leq k$.

$\forall^-$ If $t^{\forall x A}$ is an $\vec{x}$–derivation of height $\leq \alpha_0 < \alpha$ and degree $\leq k$, then $(t^{\forall x A} i)^{A[i]}$ is an $\vec{x}$–derivation of height $\leq \alpha$ and degree $\leq m$ with $m = \max(k, \text{level } \forall x A)$, if $t^{\forall x A}$ is generated by the rule ω, or of degree $\leq k$ otherwise.

We now embed our systems Z_k (i. e. arithmetic with induction restricted to Π_k^0–formulas) and hence Z into Z^∞.

Lemma 3.3.1 *Let r^B be a derivation in Z_k with free assumption variables among $\vec{x}^{\vec{A}}$ which contains $\leq m$ instances of the induction scheme all with induction formulas of level $\leq k$. Let σ be a substitution of numerals for object variables such that $\vec{A}\sigma, B\sigma$ do not contain free object variables. Then we can find an $\vec{x}^{\vec{A}\sigma}$–derivation $(r_\infty)^{B\sigma}$ in Z^∞ of height $\leq \omega^m + h$ for some $h < \omega$ and degree $\leq k$.*

Proof: First note that from any normal derivation in first–order logic we can construct a normal derivation r_0^B with the same free assumption variable $\vec{x}^{\vec{A}}$, such that in r_0^B any path has a prime formula as its minimal formula (cf p. 23). For if A is a minimal formula which is not prime we can first apply elimination rules until a prime formula is reached and later build a up again by the corresponding introduction rules.

The lemma is proved by induction on the height of the given derivation r. By the normalization theorem 2.2.1 and the note above we can assume that r is normal with prime minimal formulas. The only case which requires some argument is when r consists of two applications of $\to^-$ to an instance of the induction scheme. Then r must have the form

$$\mathrm{ax}^{A[0], \forall x(A[x]\to A[Sx])\to \forall x A[x]} s^{A[0]} (\lambda x \lambda y^{A[x]} t^{A[S[x]]}).$$

By induction hypothesis we obtain derivations

$$\begin{array}{ll} s_\infty^{A[0]} & \text{of height } \leq \omega^{m-1} + h_0 \\ t_\infty^{A[1]}[s_\infty^{A[0]}] & \text{of height } \leq \omega^{m-1} \cdot 2 + h_1, \\ t_\infty^{A[2]}[t_\infty^{A[1]}[s_\infty^{A[0]}]] & \text{of height } \leq \omega^{m-1} \cdot 3 + h_2 \end{array}$$

and so on, all of degree $\leq k$. Combining all these derivations of $A[i]$ as premisses of the ω-rule yields a derivation t_∞ of $\forall x A[x]$ of height $\leq \omega^m$ and degree $\leq k$. □

A derivation is called *convertible* if it is of the form $(\lambda x r)s$ or else $\langle r_i \rangle_{i<\omega} j$, which can be converted into $r_x[s]$ or r_j, respectively. Here $r_x[s]$ is obtained from r by substituting s for all free occurences of x in r. A derivation is called *normal* if it does not contain a convertible subderivation. Note that a derivation of degree ≤ 0 must be normal.

We want to define an operation which by repeated conversions transforms a given derivation into a normal one with the same end formula and no more assumption variables. The methods employed in sections 1 and 2 to achieve such a task have to be adapted properly in order to deal with the new situation of infinitary derivations. Here we give a particularly simple argument due to W.W. Tait [Tai65].

Lemma 3.3.2 *If r is an $\vec{x}, x^A, \vec{y}$-derivation of height $\leq \alpha$ and degree $\leq k$ and s^A is an $\vec{x}, \vec{y}$-derivation of height $\leq \beta$ and degree $\leq l$, then $r_x[s]$ is an $\vec{x}, \vec{y}$-derivation of height $\leq \beta + \alpha$ and degree $\leq \max(k, l, level\ s)$.*

This is proved by a straightforward induction on the height of r. □

Lemma 3.3.3 *For any $\vec{x}$-derivation r^A of height $\leq \alpha$ and degree $\leq k+1$ we can find an $\vec{x}$-derivation $(r^k)^A$ of height $\leq 2^\alpha$ and degree $\leq k$.*

The proof is by induction on α. The only case which requires some argument is when r is of the form ts with t of height $\leq \alpha_1 < \alpha$ and s of height $\leq \alpha_2 < \alpha$. We first consider the subcase where $t^k = \lambda x t_1$ and level $t = k+1$. Then level $s \leq k$ by the definition of level, and hence $(t_1)_x[s^k]$ has degree $\leq k$ by Lemma 3.3.2. Furthermore, also by Lemma 3.3.2, $(t_1)_x[s^k]$ has height $\leq 2^{\alpha_2} + 2^{\alpha_1} \leq 2^{\max(\alpha_2,\alpha_1)+1} \leq 2^\alpha$. Hence we can take $(ts)^k$ to be $(t_1)_x[s^k]$. If we are not in the above subcase, we can simply take $(ts)^k$ to be $t^k s^k$. This derivation clearly has height $\leq 2^\alpha$. Also it has degree $\leq k$, which can be seen as follows. If level $t \leq k$ we are done. If however level $t \geq k+2$, then t must be of the form $t_0 t_1 \dots t_m$ for some assumption variable or axiom t_0 (since r has degree $\leq k+1$). But then t^k has the form $t_0 t_1^k \dots t_m^k$ and we are done again. (To be completely precise, this last statement has to be added to the formulation of the Lemma above and proved simultaneously with it). □

As an immediate consequence we obtain

Theorem 3.3.1 *For any $\vec{x}$-derivation r^A of height $\leq \alpha$ and degree $\leq k$ we can find a normal $\vec{x}$-derivation $(r^*)^A$ of height $\leq 2_k \alpha$ (where $2_0 \alpha = \alpha, 2_{m+1}\alpha = 2^{2^\alpha_m}$).* □

3.4 Unprovable initial cases of transfinite induction

We now apply the technique of normalization for arithmetic with the ω-rule for a proof that transfinite induction up to ε_0 is underivable in Z, i. e. of

$$Z \nvdash \forall x(\forall y \prec x : Py \to Px) \to \forall x Px$$

with a predicate variable P, and that transfinite induction up to ω_{k+1} is underivable in Z_k, i. e. of

$$Z_k \nvdash \forall x(\forall x \prec x : Py \to Px) \to \forall x \prec \omega_{k+1} : Px.$$

Our proof is based on an idea of Schütte, which consists in adding a so-called *progression rule* to the infinitary systems. This rule allows to conclude Pj (where j is any numeral) from all Pi for $i \prec j$.

More precisely, we define the notion of an $\vec{x}$-derivation in $Z^\infty +$ Prog(P) of height $\leq \alpha$ and degree $\leq k$ by the inductive clauses of section 3.2 and the additional clause

Prog(P) If r_i^{Pi} are $\vec{x}$-derivations of heights $\leq \alpha_i < \alpha$ and degrees $\leq k_i \leq k$ $(i \prec j)$, then $\langle r_i^{Pi}\rangle_{i \prec j}^{Pj}$ is an $\vec{x}$-derivation of height $\leq \alpha$ and degree $\leq k$.

Since this progression rule only deals with derivations of prime formulas it does not affect the degrees of derivations. Hence the proof of normalization for Z^∞ carries over unchanged to $Z^\infty +$ Prog(P). In particular we have

Lemma 3.4.1 *For any $\vec{x}$-derivation r^A in $Z^\infty +$ Prog(P) of height $\leq \alpha$ and degree $\leq k+1$ we can find an $\vec{x}$-derivation $(r^k)^A$ in $Z^\infty +$ Prog(P) of height $\leq 2^\alpha$ and degree $\leq k$.* □

We now show that from the progression rule for P we can easily derive the progressiveness of P.

Lemma 3.4.2 *We have a normal derivation of $\forall x(\forall y \prec x : Py \to Px)$ in $Z^\infty +$ Prog(P) with height ≤ 5.*

Proof: By the ω-rule it suffices to derive $\forall y \prec j : Py \to Pj$ for any j with height ≤ 4. We argue informally. Assume $\forall y \prec j : Py$. By $\forall^-$ we have $i \prec j \to Pi$ for any i. Now for any $i \prec j$ we have $i \prec j$ as an axiom; hence Pi for any such i. An application of the progression rule yields Pj, with a derivation of height ≤ 3. Now by $\to^+$ and ω the claim follows. □

The crucial observation now is that a normal derivation of $P|\beta|$ must essentially have a height of at least β. However, to obtain the right estimates for our subsystems Z_k we cannot apply Lemma 3.4.1 down to degree 0 (i. e. to the normal form) but must stop already at degree 1. Such derivations, i. e. those of degree ≤ 1, will be called *almost normal*; they can also be analyzed easily. An almost normal derivation r in Z^∞ + Prog(P) is called a $P|\vec{\alpha}|, \neg P|\vec{\beta}|$-*refutation* if r derives a formula $\vec{A} \to B$ with $\vec{A}$ and the free assumptions in r among $P|\vec{\alpha}| :\equiv P|\alpha_1|, \dots, P|\alpha_m|$ and $\neg P|\vec{\beta}| :\equiv \neg P|\beta_1|, \dots, \neg P|\beta_n|$ and true prime formulas, and B a false prime formula or else among $\neg P|\vec{\beta}|$.

Lemma 3.4.3 *Let r be an almost normal $P|\vec{\alpha}|, \neg P|\vec{\beta}|$-refutation of height $\leq |r|$ with $\vec{\alpha}$ and $\vec{\beta}$ disjoint. Then*

$$\min \vec{\beta} \leq |r| + \#\vec{\alpha},$$

where $\#\vec{\alpha}$ denotes the number of ordinals in $\vec{\alpha}$.

Proof: By induction on α. Note that we may assume that r does not contain either ω or else $\forall^-$. Note also that r cannot be an equality axiom $\mathrm{ax}^{P|\gamma|,|\gamma|=|\delta| \to P|\delta|}$ with $\gamma = \delta$ true, since we have assumed that $\vec{\alpha}$ and $\vec{\beta}$ are disjoint. We distinguish cases according to the last rule in r.

Case $\to^+$: By our definition of refutations the claim follows immediately from the induction hypothesis.

Case $\to^-$: Then $r \equiv t^{A \to (\vec{A} \to B)} s^A$. If A is a true prime formula, the claim follows from the induction hypothesis for t. If A is a false prime formula, the claim follows from the induction hypothesis for s. If A is $\neg\neg P|\gamma|$ (and hence $t \equiv \mathrm{ax}^{\forall x(\neg\neg Px \to Px)}|\gamma|$), then since the level of $\neg\neg P|\gamma|$ is 2 the derivation $s^{\neg\neg P|\gamma|}$ must end with an introduction rule, i. e. $s \equiv \lambda x^{\neg P|\gamma|} s_0^{\perp}$ (for otherwise, since no axiom contains some $\neg\neg Pr_0$ as a strictly positive subformula, we would get a contradiction against the assumption that r has degree ≤ 1). The claim now follows from the induction hypothesis for s_0. The only remaining case is when A is $P|\gamma|$. Then t is an almost normal $P|\gamma|, P|\vec{\alpha}|, \neg P|\vec{\beta}|$ -refutation and s is an almost normal $P|\vec{\alpha}|, \neg P|\vec{\beta}|, \neg P|\gamma|$ -refutation. We may assume that γ is not among $\vec{\alpha}$, since otherwise the claim follows immediately from the induction hypothesis for t. Hence we have by the induction hypothesis for t

$$\min \vec{\beta} \leq |t| + \#\vec{\alpha} + 1 \leq |r| + \#\vec{\alpha}.$$

Case Prog(P). Then $r \equiv \langle r_\delta^{P|\delta|} \rangle^{P|\gamma|}_{\delta<\gamma}$. By induction hypothesis, since r_δ is a $P|\vec{\alpha}|, \neg P|\vec{\beta}|, \neg P|\delta|$ -refutation, we have for all $\delta < \gamma$

$$\min(\vec{\beta}, \delta) \leq |r_\delta| + \#\vec{\alpha} < |r| + \#\vec{\alpha}$$

ans hence

$$\min(\vec{\beta}, \gamma) \leq |r| + \#\vec{\alpha}. \quad \square$$

Now we can show

Theorem 3.4.1 *Transfinite induction up to ε_0 is underivable in Z, i. e.*

$$Z \nvdash \forall x(\forall y \prec x : Py \rightarrow Px) \rightarrow \forall x Px$$

with a predicate variable P, and that transfinite induction up to ω_{k+1} is underivable in Z_k, i. e.

$$Z_k \nvdash \forall x(\forall x \prec x : Py \rightarrow Px) \rightarrow \forall x \prec \omega_{k+1} : Px.$$

Proof: We restrict ourselves to the second part. So assume that transfinite induction up to ω_{k+1} is derivable in Z_k. Then by the embedding of Z_k into Z^∞ (Lemma 3.3.1) and the normal derivability of the progressiveness of P in Z^∞ + Prog(P) with finite height (Lemma 3.4.2) we can conclude that $\forall x \prec \omega_{k+1} : Px$ is derivable in Z^∞ + Prog(P) with height $< \omega^m + h$ for some $m, h < \omega$ and degree $\leq k$. Now $k-1$ applications of Lemma 3.4.1 yield a derivation of the same formula $\forall x \prec \omega_{k+1} : Px$ in Z^∞ + Prog(P) with height $\leq \gamma \leq 2_{k-1}(\omega^m + h) < \omega_{k+1}$ and degree ≤ 1, hence also a derivation of $P|\gamma+1|$ in Z^∞ + Prog(P) with height $\leq \gamma$ and degree ≤ 1. But this contradicts Lemma 3.4.3.$\square$

3.5 Normalization for finitary arithmetic is impossible

The normalization theorem for first-order logic applied to arithmetic Z is not particularly useful since we may have used in our derivation induction axioms of arbitrary complexity. Hence it is tempting to first eliminate the induction scheme in favour of an induction rule allowing to conclude $\forall x A[x]$ from a derivation of $A[0]$ and a derivation of $A[Sx]$ with an additional assumption $A[x]$ to be cancelled at this point (note that this rule is equivalent to the induction scheme), and then to try to normalize the resulting derivation in the new system Z with the induction rule. We will apply our results from section 3.4 to show that even a very weak form of the normalization theorem cannot hold in Z with the induction rule.

Theorem 3.5.1 *The following weak form of a normalization theorem for Z with the induction rule is false: For any $\vec{x}^{\vec{A}}$-derivation r^B with $\vec{A}, B$ Π^0_l-formulas there is an $\vec{x}^{\vec{A}}$-derivation $(r^*)^B$ containing only Π^0_k-formulas, with k depending only on l.*

Proof: Assume that such a normalization theorem would hold. Consider the Π^0_3-formula

$$\forall x(\forall y \prec x : Py \to Px) \to \forall x \prec \omega_{n+1} : Px$$

expressing transfinite induction up to ω_{n+1}. By Theorem 3.2.1 it is derivable in Z. Hence there exists a derivation of the same formula containing only Π^0_k-formulas, for some k independent of n. Hence Z_k derives transfinite induction up to ω_{n+1} for any n. But this clearly contradicts Theorem 3.4.1. □æ

Bibliography

[Bac55] Heinz Bachmann. *Transfinite Zahlen*. Springer, Berlin, 1955.

[Bar84] Hendrik Pieter Barendregt. *The Lambda Calculus*. North-Holland Publishing Company, Amsterdam, second edition, 1984.

[Chu41] Alonzo Church. *The calculi of lambda-conversion*. Annals of Math. Studies No.6, Princeton, 1941.

[Dil68] Justus Diller. Zur Berechenbarkeit primitiv rekursiver Funktionale endlicher Typen. In Kurt Schütte, editor, *Contributions to Mathematical Logic*, pages 109–120, North-Holland Publishing Company, Amsterdam, 1968.

[Gan80a] Robin.O. Gandy. An early proof of normalization. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 453–455. Academic Press, 1980.

[Gan80b] Robin.O. Gandy. Proofs of strong of normalization. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 457–477. Academic Press, 1980.

[Gen43] Gerhard Gentzen. Beweisbarkeit und Unbeweisbarkeit von Anfangsfällen der transfiniten Induktion in der reinen Zahlentheorie. *Mathematische Annalen*, 119:140–161, 1943.

[Gir87] Jean Yves Girard. *Proof Theory and Logical Complexity*. Bibliopolis, Napoli, 1987.

[How80a] William.A. Howard. The formulae-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry:*

Essays on Combinatory Logic, Lambda Calculus and Formalism, pages 479–490, Academic Press, 1980.

[How80b] William.A. Howard. Ordinal analysis of terms of finite type. *The Journal of Symbolic Logic*, 45(3):493–504, 1980.

[Pra65] Dag Prawitz. *Natural Deduction*. Volume 3 of *Acta Universitatis Stockholmiensis. Stockholm Studies in Philosophy*, Almqvist & Wiksell, Stockholm, 1965.

[San67] Luis E. Sanchis. Functionals defined by recursion. *Notre Dame Journal of Formal Logic*, 8:161–174, 1967.

[Sch77a] Kurt Schütte. *Proof Theory*. Springer, Berlin, 1977.

[Sch77b] Helmut Schwichtenberg. Proof theory: some applications of cut-elimination. In Jon Barwise, editor, *Handbook of Mathematical Logic*, chapter Proof Theory and Constructive Mathematics, pages 867–895, North-Holland Publishing Company, Amsterdam, 1977.

[Sch82] Helmut Schwichtenberg. Complexity of normalization in the pure typed λ–calculus. In Anne S. Troelstra and Dirk van Dalen, editors, *The L.E.J. Brouwer Centenary Symposium. Proceedings of the Conference hald in Noordwijkerhout, 8-13 June, 1981*, pages 453–458, North-Holland Publishing Company, Amsterdam, 1982.

[Sch86] Helmut Schwichtenberg. A normal form for natural deductions in a type theory with realizing terms. In V. Michele Abrusci and Ettore Casari, editors, *Atti del Congresso* Logica e Filosofia della Scienza, oggi, *San Gimignano, 7-11 dicembre 1983, Vol.1-Logica*, pages 95–138, CLUEB, Bologna, 1986.

[Sch88] Helmut Schwichtenberg. LCF with realizing terms: a framework for the development and verification of programs. July 1988. Unpublished Manuscript.

[Sta79] Richard Statman. The typed λ–calculus is not elementary recursive. *Theoretical Computer Science*, 9:73–81, 1979.

[Tai65] W.W. Tait. Infinitely long terms of transfinite type. In J. Crossley and M. Dummett, editors, *Formal Systems and Recursive Functions*, pages 176–185, North-Holland Publishing Company, Amsterdam, 1965.

[Tak87] Gaisi Takeuti. *Proof Theory.* North-Holland Publishing Company, Amsterdam, second edition, 1987.

[Tro73] Anne Troelstra, editor. *Metamathematical Investigations of Intuitionistic Arithmetic and Analysis.* Volume 344 of *Lecture Notes in Mathematics*, Springer, 1973.

[TvD88] Anne S. Troelstra and Dirk van Dalen. *Constructivism in Mathematics. An Introduction.* Volume 121, 123 of *Studies in Logic and the Foundations of Mathematics*, North-Holland Publishing Company, Amsterdam, 1988.

LCF with realizing terms: a framework for the development and verification of programs

Helmut Schwichtenberg*
Mathematisches Institut der Universität München

July 25, 1988

A continuous functional in the sense of Scott and Ersov (cf. [14], [5], [12], [16])is computable iff it is given by a recursive enumeration of its finite approximations. In section 1 we describe a functional programming language L which may be used to define computable functionals and give an operational semantics for it. Parallel computation is necessary because non-strict functionals are allowed. The rules of the operational semantics for L are then extended into a formal system $\mathcal{T}$ which is also an extension of Scott's LCF introduced in [13]. $\mathcal{T}$ can be used to prove that programs in L meet their specifications. The specification language contains (as in [15]) a constructive existential quantifier $\exists^*$ in addition to the usual classical quantifier $\exists$ (defined by $\neg\forall\neg$). This provides a useful means to express in certain situations exactly the intended computational content of a formula[1]. E.g. the subtype construct in [4, p.32], whose purpose is to *to provide a mechanism for hiding information*, becomes superfluous here. We also give two examples of programs: In section 3 for the Warshall algorithm, which decides reachability in finite graphs, and in section 4 for an algorithm derived from a proof of a version of the intermediate value

*Thanks are due to Carnegie-Mellon-University (Pittsburgh, Pennsylvania, USA) for its hospitality during my visit there in the academic year 1987/88, and to the Stiftung Volkswagenwerk for a grant which made this visit possible. I further want to thank Steve Brookes, Daniel Leivant, Frank Pfenning, Dana Scott and Rick Statman for helpful discussions.

[1]The distinction made in [10] between *the propositions that have a "computational informative" contents and the propositions that only have a "logical" contents* seems to serve a similar purpose.

theorem in constructive real analysis, which may be specialized to compute the squareroot of 2. It is discussed how one can derive in $\mathcal{T}$ that these programs meet their specifications.

1 Operational semantics for a functional programming language

We use simple types built up from **nat** and **bool** by $\rho \to \sigma$. *Expressions* E are built up by application from

1. typed variables $X : \rho, Y : \rho, \dots$
2. constants k_A for all finite approximations A (see [16])
3. some other constants, e.g.[2]
   - $+1, -1 : \mathbf{nat} \to \mathbf{nat}$
   - $\mathbf{Zero} : \mathbf{nat} \to \mathbf{bool}$
   - $=, \leq, < : \mathbf{nat} \to (\mathbf{nat} \to \mathbf{bool})$

Expressions of type **bool** are denoted by B. *Programs* are built up from assignments $Y := E$ (where the variable Y may occur in E) and **skip** by

- $P_1; \dots ; P_n$
- **if** B **then** P **else** Q **fi**
- **for** $X = 1$ **to** Z **do** P **od**, where $Z \notin var(P)$
- **while** $X = 0$ **do** P **od**

Here $var(P)$ should consist of all variables which P may either read to get an input or else write into to produce output. More precisely, we define

- $var(Y := E) = var(E) \cup \{Y\}$

[2]Note that constants like **true, false:bool** and **not: bool→bool** need not be mentioned since they are given by finite approximations.

- $var(\textbf{skip}) = \emptyset$
- $var(P_1; \ldots ; P_n) = var(P_1) \cup \cdots \cup var(P_n)$
- $var(\textbf{if } B \textbf{ then } P \textbf{ else } Q \textbf{ od}) = var(B) \cup var(P) \cup var(Q)$
- $var(\textbf{for } X = 1 \textbf{ to } Z \textbf{ do } P \textbf{ od}) = (var(P) \cup \{Z\}) \setminus \{X\}$
- $var(\textbf{while } X = 0 \textbf{ do } P \textbf{ od}) = var(P) \cup \{X\}$

We now give the rules of our operational semantics, in the style of Hoare-logics. However, we only have to deal here with Hoare-triples having equations as pre- and postconditions[3].

$\boxed{\textbf{Thinning rules}}$ Let $X = var(P)$.

$$\frac{y \sqsupseteq x \qquad \{X = x\}P\{X = x'\} \qquad x' \sqsupseteq y'}{\{X = y\}P\{X = y'\}}$$

$$\{X = \bot\}P\{X = \bot\}$$

$$\frac{\{X = x\}P\{X = x'\}}{\{X = x, Y = y\}P\{X = x', Y = y\}}$$

where in the last rule it is assumed that the sequence Y of variables contains no variable from $var(P)$[4].

$\boxed{Y := E}$ Let X be minimal such that $var(E) \subseteq \{X, Y\}$.

$$\{X = x, Y = y\}Y := E\{X = x, Y = E(X, Y/x, y)\}$$

$\boxed{\textbf{skip}}$ This rule allows just to write down **skip** considered as a Hoare-triple with empty pre- and postconditions.

[3]Such Hoare-triples have also received special attention in recent work of S. Brookes; see [2].

[4]Here and later we use $X, Y, \ldots$ to denote single variables as well as sequences of variables.

$\boxed{P;Q}$ Let $X = var(P) \cup var(Q)$.

$$\frac{\{X=x\}P\{X=x'\} \qquad \{X=x'\}Q\{X=x''\}}{\{X=x\}P;Q\{X=x''\}}$$

$\boxed{\textbf{if } B \textbf{ then } P \textbf{ else } Q \textbf{ fi}}$ Denote this program by R, and let $X = var(R)$.

$$\frac{B(X/x) = \textbf{true} \qquad \{X=x\}P\{X=x'\}}{\{X=x\}R\{X=x'\}}$$

and similarly for **false**.

$$\frac{\{X=x\}P\{X=x'\} \qquad \{X=x\}Q\{X=x'\}}{\{X=x\}R\{X=x'\}}$$

$\boxed{\textbf{for } X=1 \textbf{ to } Z \textbf{ do } P \textbf{ od}}$ Denote this program by Q, and note that $Z \notin var(P)$. Let Y be minimal such that $var(P) \subseteq \{X,Y\}$.

$$\{Z=0, Y=y\}Q\{Z=0, Y=y\}$$

$$\frac{\{Z=z, Y=y\}Q\{Z=z, Y=y'\} \quad \{X=z+1, Y=y'\}P\{X=x', Y=y''\} \quad z \neq \bot}{\{Z=z+1, Y=y\}Q\{Z=z+1, Y=y''\}}$$

$\boxed{\textbf{while } X=0 \textbf{ do } P \textbf{ od}}$ Denote this program by Q. Let Y be minimal such that $var(P) \subseteq \{X,Y\}$. We use an auxiliary relation constant P^*.

$$P^*xyxy$$

$$\frac{P^*xy0y' \qquad \{X=0, Y=y'\}P\{X=x', Y=y''\}}{P^*xyx'y''}$$

$$\frac{P^*xyx'y' \qquad x' \neq 0, \bot}{\{X=x, Y=y\}Q\{X=x', Y=y'\}}$$

To simplify our notation, we often leave out preconditions of the form $X = \bot$. We also allow ourselves to leave out any of the postconditions. So asserting the derivability of $\{X = x\}P\{Y = y\}$ means that for some x', we can derive $\{X = x, Y = \bot\}P\{X = x', Y = y\}$

Lemma 1 (Uniqueness) *If* $\{X = x\}P\{Y = y\}$ *and* $\{X = x\}P\{Y = y'\}$ *are derivable with* $Y : \mathbf{nat}$ *and* $y, y' \neq \bot$, *then* $y = y'$.

For the proof one has to generalize the claim sightly to cover the case that Y is of a higher type.

By the Uniqueness Lemma, any program P defines with respect to $X \subseteq var(P)$ taken as input variables and $Y \in var(P)$ taken as output variable a functional. It can be shown that the functionals definable in L in this way are exactly the computable ones. For the proof one may use either Feferman's (see [6]) or else Plotkin's (see [12]) characterization of the computable functionals by means of schemata.

The fragment of L obtained by leaving out **while**-loops defines a proper subclass of the computable functionals, which may be called *partial primitive recursive* functionals. These are closely related to the Gödel primitive recursive functionals of [7]: A functional is partial primitive recursive iff it is the restriction of a Gödel primitive recursive functional to a Gödel primitive recursive domain.

2 Formal verification of programs

In our specification language we distinguish between the classical existential quantifier $\exists$, defined by $\neg\forall\neg$, and a constructive existential quantifier denoted by $\exists^*$; $\vee^*$ can then be defined by

$$A \vee^* B := \exists^*((x = \mathbf{true} \rightarrow A) \wedge (x = \mathbf{false} \rightarrow B)).$$

Formulae without $\exists^*$ are called *negative* and their proofs do not have any computational content. More generally, this also holds for formulae which contain $\exists^*$ only on left-hand-sides of implications, the so-called *Harrop-formulae*. Note that $\neg A$ is considered as defined by $A \rightarrow \mathbf{false}$.

A formula containing $\exists^*$ can be considered as posing a problem or a task, which may be fulfilled by providing certain functionals realizing it in the sense of [8]. E.g., $\forall x \exists^* y . x < y$ requires a function, $\forall x \exists^* y \exists^* z . x < y < z$ requires two functions, and $(\forall x \exists^* y . x < y) \rightarrow \exists^* z . 0 < z$ requires a functional of type $(\mathbf{nat} \rightarrow \mathbf{nat}) \rightarrow \mathbf{nat}$.

Generally, for any formula A we define a finite sequence $\rho_1, \ldots, \rho_n := \rho$ of types such that functionals $\xi_1, \ldots, \xi_n$ of these types are needed to realize A. The definition is as follows. To a prime formula there belongs the empty sequence of types. If $\vec{\rho}$ are the types of A and $\vec{\sigma} = \sigma_1, \ldots, \sigma_m$ are the types of B, then

- $\vec{\rho}, \vec{\sigma}$ are the types of $A \wedge B$,
- $\vec{\rho} \rightarrow \sigma_1, \ldots, \vec{\rho} \rightarrow \sigma_m$ are the types of $A \rightarrow B$,
- $\rho \rightarrow \sigma_1, \ldots, \rho \rightarrow \sigma_m$ are the types of $\forall \xi : \rho . B$ and
- $\rho, \vec{\sigma}$ are the types of $\exists^* \xi : \rho . B$.

So, in particular, to any Harrop-formula A there belongs the empty sequence of types.

To say that a program P meets its specification A then means that P can be considered as defining (see p.5) a sequence $\xi_1, \ldots, \xi_n$ of functionals of the types $\rho_1, \ldots, \rho_n$ associated to A as above, and that $\xi_1, \ldots, \xi_n$ realize A (written $\xi_1, \ldots, \xi_n \in A$) in the sense of the following definition.

$$\vec{\xi}, \vec{\zeta} \in A \wedge B \quad :\leftrightarrow \quad \vec{\xi} \in A \ \wedge \ \vec{\zeta} \in B$$

$$\zeta_1, \ldots, \zeta_m \in A \rightarrow B \quad :\leftrightarrow \quad \forall \vec{\xi} (\vec{\xi} \in A \ \rightarrow \ \zeta_1 \vec{\xi}, \ldots, \zeta_m \vec{\xi} \in B)$$

$$\xi_1, \ldots, \xi_m \in \forall \xi . A \quad :\leftrightarrow \quad \forall \xi ((\zeta_1 \xi, \ldots, \zeta_m \xi \in A)$$

$$\vec{\zeta} \in \exists^* \xi . A \quad :\leftrightarrow \quad \xi, \vec{\zeta} \in A$$

So any *judgement*[5] $\xi_1 \ldots, \xi_n \in A$ with A a formula possibly containing the constructive existential quantifier $\exists^*$ can be replaced by an ordinary negative formula and hence, in principle, it is not necessary to consider judgements. However, in the context of program verification it is very useful to have the suggestive notation of judgements available.

Note that the distinction between $\exists$ and $\exists^*$ can also be used to obtain the effect of the subtype construct in [4, p.32]. As it is stated there, the main purpose

[5]This terminology is due to Weyl and was taken up by Martin-Löf; see [9]

of the subtype construct is to *provide a mechanism for hiding information to simplify computation.* Now this can obviously be achieved by considering all $\xi : \mathbf{nat} \rightarrow \mathbf{nat}$ satisfying $\exists x : \mathbf{nat}.\xi(x) = 0$ instead of $\exists^* x : \mathbf{nat}.\xi(x) = 0$.

Formal verification of programs requires the setup of a formal system $\mathcal{T}$ which allows the derivation of certain valid judgements.We do not go into the details of $\mathcal{T}$ here but only note some of the essential points. $\mathcal{T}$ has variables $x, y, z, \ldots$ ranging over finite approximations as well as variables $\xi, \zeta, \eta, \ldots$ ranging over ideals of finite approximations(i.e.continuous functionals), both for all simple types. So $\mathcal{T}$ is essentially a second order system. However, the existence axioms for the second order variables $\xi, \zeta, \eta, \ldots$ are weak and require only the existence of the computable functionals. The basic ingredients of $\mathcal{T}$ are the rules given above for our operational semantics. They are considered as a means to derive prime formulae. We also allow formal induction over **nat** as a rule in $\mathcal{T}$, as well as a version of Scott's induction on $\sqsubseteq$ introduced in [13]; hence $\mathcal{T}$ can be considered as an extension of Scott's LCF.

Using standard techniques from proof theory[6] it can be shown that $\mathcal{T}$ is a conservative extension of classical arithmetic Z.

3 The Warshall algorithm

We have choosen the Warshall algorithm as an example because it has been dealt with in the literature in a number of places (see Broy and Pepper [3], Pfenning [11]), and so our treatment can be compared with theirs.

Given a finite directed graph and two nodes x, y in it, the Warshall algorithm decides whether there is a path p in the graph connecting x and y. Furthermore, it provides such a path if there is any.

We first write out a formal specification of the algorithm. In order to do that conveniently, we extend our type system by allowing cartesian products $\rho \times \sigma$ and list types **list**ρ. *Expressions E* may now also contain the constants [7]

- **pair** : $\rho \rightarrow (\sigma \rightarrow \rho \times \sigma)$
- **fst** : $\rho \times \sigma \rightarrow \rho$

[6] Either by defining a model for $\mathcal{T}$ within Z, or else directly by normalization as in [15].
[7] Here and later we use a notation derived from Common Lisp; see [17]

- $\mathbf{snd} : \rho \times \sigma \to \sigma$
- $\mathbf{nil} : \mathbf{list}\rho$
- $\mathbf{cons} : \rho \times \mathbf{list}\rho \to \mathbf{list}\rho$
- $\mathbf{car} : \mathbf{list}\rho \to \rho$
- $\mathbf{cdr} : \mathbf{list}\rho \to \mathbf{list}\rho$

We also assume that we have the following constants available.

$$\mathbf{assoc} : \rho \times \mathbf{list}(\rho \times \sigma) \to \sigma$$

$$\mathbf{append} : \mathbf{list}\rho \times \mathbf{list}\rho \to \mathbf{list}\rho$$

$$\mathbf{length} : \mathbf{list}\rho \to \mathbf{nat}$$

$$\mathbf{elt} : \mathbf{list}\rho \times \mathbf{nat} \to \rho$$

and that our formal system derives the obvious (conditional) equations for these, e.g.[8]

$$\mathbf{assoc}(x, \mathbf{cons}(\mathbf{pair}(x, y), l)) = y$$

$$x \neq x' \to \mathbf{assoc}(x, \mathbf{cons}(\mathbf{pair}(x', y,), l)) = \mathbf{assoc}(x, l)$$

$$\mathbf{assoc}(x, \mathbf{nil}) = \mathbf{nil}$$

$$\mathbf{append}(\mathbf{nil}, q) = q$$

$$\mathbf{append}(\mathbf{cons}(x, p), q) = \mathbf{cons}(x, \mathbf{append}(p, q))$$

In our specification, we assume that the nodes x, y of the graph are given by natural numbers $< n$, and that the result of the algorithm consists of an association list $a : \mathbf{list}((\mathbf{nat} \times \mathbf{nat}) \times \mathbf{list}\ \mathbf{nat})$ with the following property. If there is a path p in the graph leading from x to y, then a assigns something to $\mathbf{pair}(x, y)$, and if a assigns something to $\mathbf{pair}(x, y)$, then this is a path p leading from x to y.

[8]Note that we may easily write out programs (without while-loops) which define assoc, append etc. and then treat formulas containing such defined functions as abbreviations for other formulas which contain instead Hoare-triples for these programs. In this setup the equations above become provable.

It is convenient to assume that the edges of the graph are given in the form of an association list a : **list**((**nat** $\times$ **nat**) $\times$ **list nat**), which assigns to any x, y connected in the graph the list (x, y) of length 2.

Our specification now reads as follows:

$$\begin{array}{ll} & \forall e, n \exists^* a \forall x, y, p(\mathbf{init}(e, n) \wedge x, y < n \rightarrow \\ (1) & \quad (\mathbf{path}(e, x, y, p) \rightarrow \mathbf{assoc}(\mathbf{pair}(x, y), a) \neq \mathbf{nil}) \wedge \\ & \quad (\mathbf{assoc}(\mathbf{pair}(x, y)), a) \neq \mathbf{nil} \rightarrow \mathbf{path}(e, x, y, \mathbf{assoc}(\mathbf{pair}(x, y), a)))) \end{array}$$

where

$$\begin{array}{rcl} \mathbf{init}(e, n) & := & \forall i < \mathbf{length}(e)[\mathbf{length}(\mathbf{snd}(\mathbf{elt}(e, i))) = 2 \wedge \\ & & \mathbf{fst}(\mathbf{fst}(\mathbf{elt}(e, i))) = \mathbf{car}(\mathbf{snd}(\mathbf{elt}(e, i))) < n \wedge \\ & & \mathbf{snd}(\mathbf{fst}(\mathbf{elt}(e, i))) = \mathbf{car}(\mathbf{cdr}(\mathbf{snd}(\mathbf{elt}(e, i)))) < n] \\ \mathbf{path}(e, x, y, p) & := & \mathbf{elt}(p, 0) = x \wedge \\ & & \mathbf{elt}(p, \mathbf{length}(p) - 1) = y \wedge \\ & & \forall i < \mathbf{length}(p) - 1(\mathbf{assoc}(\mathbf{pair}(\mathbf{elt}(p, i), \mathbf{elt}(p, i + 1)), e) \neq \mathbf{nil}) \end{array}$$

An obvious way to give a constructive proof of (1) and hence an algorithm yielding a from e, n would be to introduce an additional parameter m and use induction on $m \leq n$ to prove

$$\begin{array}{ll} & \forall e, n, m \exists^* a \forall x, y, p(\mathbf{init}(e, n) \wedge x, y < n \wedge m \leq n \rightarrow \\ (2) & \quad (\mathbf{path}(e, m, x, y, p) \rightarrow \mathbf{assoc}(\mathbf{pair}(x, y), a) \neq \mathbf{nil}) \wedge \\ & \quad (\mathbf{assoc}(\mathbf{pair}(x, y)), a) \neq \mathbf{nil} \rightarrow \mathbf{path}(e, m, x, y, \mathbf{assoc}(\mathbf{pair}(x, y), a)))) \end{array}$$

where **init**(e, n) is as above and

$$\begin{array}{rcl} \mathbf{path}(e, m, x, y, p) & := & \mathbf{elt}(p, 0) = x \wedge \\ & & \mathbf{elt}(p, \mathbf{length}(p) - 1) = y \wedge \\ & & \forall i < \mathbf{length}(p) - 1(\mathbf{assoc}(\mathbf{pair}(\mathbf{elt}(p, i), \mathbf{elt}(p, i + 1)), e) \neq \mathbf{nil}) \\ & & \forall i < \mathbf{length}(p) - 2(\mathbf{elt}(p, i + 1) < m) \end{array}$$

In the induction step a_{m+1} is obtained from a_m by appending a certain increment i to it. This increment is again an association list. For any pair $x, y < n$ it has an entry iff a_m does not already assign a connecting path to x, y, but a_m does assign a path xpm to x, m and also a path mqy to m, y. Then i assigns the path $xpqm$ to x, y.

Note that the use of association lists to store the previous results is essential for the effectiveness of the algorithm[9]. If instead we would have used recursive calls of the algorithm itself, then the fact that for any x, y, m we need two such recursive calls would lead to an exponentially growing number of recomputations.

Our program P_W for the Warshall algorithm now reads as follows:

```
A := E;
for Z = 1 to M do
   I := nil;
   for X = 1 to N do
      for Y = 1 to N do
         if assoc(pair(X, Y), A) = nil
           and assoc(pair(X, Z), A) ≠ nil
           and assoc(pair(Z, Y), A) ≠ nil
         then I :=cons(pair(pair(X, Y),
                             append(assoc(pair(X, Z), A)
                                     cdr(assoc(pair(Z, Y), A)))),
                             I)
         else skip
         fi
      od
   od;
   A := append(I, A)
od
```

If we abbreviate the specification (2) above by $\forall e, n, m \exists^* a S(e, n, m, a)$, then P_W, considered as a functional mapping values of E, N, M into a value of A, realizes

[9]This has also been stressed in [11].

(2) in the sense of section 2, i.e. we can derive in $\mathcal{T}$

$$\{E = e, N = n, M = m\} P_W \{A = a\} \to S(e, n, m, a),$$

using formal induction on m.

4 An algorithm for the intermediate value theorem

We have choosen this example since it involves a treatment of some basic concepts from constructive analysis in the style of Bishop [1], and also makes use of the fact that we have a higher order language available.

Let $\bar{f}$ be a monotonic, uniformly continuous real function on $[0, 1]$ such that $\bar{f}(0) < 0 < \bar{f}(1)$. To simplify matters we assume that $\bar{f}$ happens to map rationals into rationals; so $\bar{f}$ can be restricted to a function $f : Q \cap [0, 1] \to Q$. We want to construct a real a such that $\bar{f}(a) = 0$. To achieve this, we construct sequences a_n, b_n of rationals such that

$$f(a_n) \le 0 < f(b_n)$$

$$b_n - a_n = \frac{1}{2^n}$$

$$0 = a_0 \le a_n \le a_{n+1} < b_{n+1} \le b_n \le b_0 = 1$$

Then obviously (a_n) is a Cauchy sequence with Cauchy modulus $M(\epsilon) :=$ *least m such that* $\frac{1}{2^m} \le \epsilon$. Hence we have a real which is a zero of $\bar{f}$.

We now want to write out the corresponding programs and the specifications they are supposed to satisfy. To do this we have to be somewhat more explicit. Let $Q := (\mathbf{nat} \times \mathbf{bool}) \times \mathbf{nat}$ be the type of rationals; for $a : Q$, we write $a \in Q$ to mean $a \neq \bot \land \mathbf{snd}(a) \neq 0$ and $a \in Q^+$ to mean $a \in Q \land \mathbf{snd}(\mathbf{fst}(a)) = \mathbf{true}$. Assume

$$f(0), f(1) \neq \bot \;\land\; f(0) \le 0 < f(1) \tag{1}$$

$$\forall a, b \in Q \cap [0, 1] (f(a), f(b) \neq \bot \land a \le b \to f(a) \le f(b) \in Q) \tag{2}$$

$$\forall \epsilon \in Q^+ (\omega(\epsilon) \neq \bot \to \omega(\epsilon) \in Q^+) \tag{3}$$

$$\forall \epsilon \in Q^+ \forall a, b \in Q \cap [0, 1] (\omega(\epsilon) \neq \bot \land f(a), f(b) \neq \bot \land |a - b| \le \omega(\epsilon) \to |f(a) - f(b)| \le \epsilon) \tag{4}$$

To construct the sequences a_n, b_n we use the program P_{seq}:

```
$A$ := 0;
$B$ := 1;
for $Y = 1$ to $N$ do
    $C := A + \frac{1}{2}(B - A)$;
    $Z$ := app($F, C$);
    if $0 < Z$
    then $B := C$
    else $A := C$
    fi
od
```

Then we can derive, using induction on n,

$$\{F = f, N = n\} P_{seq} \{A = a_n, B = b_n\} \wedge a_n, b_n \neq \perp \wedge (1,2) \rightarrow$$
$$f(a_n), f(b_n) \neq \perp \wedge$$
$$f(a_n) \leq 0 < f(b_n) \wedge$$
$$b_n - a_n = \frac{1}{2^n} \wedge$$
$$0 \leq a_n < b_n \leq 1$$

So in this sense P_{seq} computes a zero for $\overline{f}$.

Furthermore, we may use P_{seq} to obtain, given $\epsilon \in Q^+$, an approximation a_n to this zero of $\overline{f}$ such that $|f(a_n)| \leq \epsilon$.To achieve this, we first construct a program P_{min} which, given $\epsilon \in Q^+$, computes an n such that $\frac{1}{2^n} \leq \omega(\epsilon)$:

```
Delta := app(Omega, Epsilon);
$Q$ :=snd(Delta);
$P$ :=fst(fst(Delta));
for $X = 0$ to $Q$ do
    if $Q \leq 2^X * P$
    then $N := X$
    else skip
od
```

Then for P_{min} we can derive

$\{\text{Epsilon} = \epsilon, \text{Omega} = \omega\} P_{min} \{N = n\}$
$\wedge\ \epsilon \in Q^+$
$\wedge\ \omega(\epsilon) \neq \perp$
$\wedge\ (3)$
$\rightarrow\ n \neq \perp$
$\quad \wedge\ \frac{1}{2^n} \leq \omega(\epsilon)$

Finally we have, as required,

$\{F = f, \text{Epsilon} = \epsilon, \text{Omega} = \omega\} P_{min}; P_{seq} \{A = a, B = b\}$
$\wedge\ \epsilon \in Q^+$
$\wedge\ \omega(\epsilon) \neq \perp$
$\wedge\ a, b \neq \perp$
$\wedge\ (1-4)$
$\rightarrow\ |f(a)| \leq \epsilon.$

References

[1] E.Bishop *Foundations of Constructive Analysis*. McGraw Hill, New York, 1967.

[2] S. Brookes. *Semantics of programming languages*. Course Notes, Carnegie-Mellon-University, Pittsburgh, 1988.

[3] M. Broy and P. Pepper. *Program development as a formal activity*. IEEE Transactions on Software Engineering, SE-7(1), pp.44-67, 1977.

[4] R. Constable et al. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, Englewood Cliffs, New Jersey, 1986.

[5] Yu. Ersov. *The model C of partial continuous functionals*. In *Logic colloquium 76* (ed. R. Gandy and M. Hyland), North Holland, Amsterdam, pp.455-467, 1977.

[6] S. Feferman. *Generating schemes for partial recursively continuous functionals (Summary)*. In *Colloque International de Logique, Clermont-Ferrand 1975*. CNRS, Paris, pp.191-198, 1977

[7] K. Gödel. *Über eine noch nicht benützte Erweiterung des finiten Standpunkts*. Dialectica 12, pp.280-287, 1958.

[8] G. Kreisel. *Interpretation of analysis by means of constructive functionals of finite types*. In *Constructivity in mathematics* (ed. A.Heyting), North Holland, Amsterdam, pp.101-128, 1959.

[9] P. Martin-Löf. *Constructive mathematics and computer programming*. In *Logic, Methodology and the Philosophy of Science VI*. North-Holland, Amsterdam, pp.153-175, 1980.

[10] C. Paulin-Mohring. *Extracting F_ω's programs from proofs in the calculus of constructions*. Extended Abstract, INRIA and LIENS, 1988.

[11] F. Pfenning. *Program development through proof transformation*. Ergo-Report 87-047, Carnegie-Mellon-University, Pittsburgh, 1987. To appear in a volume *Logic and Computation* of the AMS series *Contemporary Mathematics*, ed. W.Sieg.

[12] G. Plotkin. *LCF considered as a programming language*. Theor. Comp. Science 5, pp.223-255, 1977.

[13] D. Scott. *A type theoretical alternative to ISWIM, CUCH, OWHY*. Manuscript, Oxford, October 1969.

[14] D. Scott. *Domains for denotational semantics*. In *Automata, languages and programming* (ed. M. Nielsen and E.M. Schmidt), Springer Lecture Notes in Computer Science 150, pp.577- 613, 1982.

[15] H. Schwichtenberg. *A normal form for natural deductions in a type theory with realizing terms* In *Atti del Congresso Logica e Filosofia della Scienza, oggi. San Gimignano, 7-11 dicembre 1983. Vol.I-Logica*. CLUEB, Bologna, pp.95-138, 1986.

[16] H. Schwichtenberg. *Eine Normalform für endliche Approximationen von partiellen stetigen Funktionalen*. In *Logik und Grundlagenforschung. Festschrift zum 100. Geburtstag von Heinrich Scholz* (ed. J. Diller), Aschendorff, Münster, pp.89-95, 1986.

[17] G. Steele Jr. *Common Lisp*. Digital Press, 1984.